王静文 吴晓艺 编著

密码编码与信息安全

——C++实践



清华大学出版社

密码编码与信息安全
——C++实践

王静文　吴晓艺　编著

清华大学出版社
北　京

内 容 简 介

本书主要介绍了密码编码学与信息安全的常用算法所涉及的理论，并介绍了使用 C++ 语言实现这些算法的基本过程与具体实现。本书涵盖了古典密码、对称密钥算法、公钥算法、散列函数和数字签名等几部分内容，并在每章最后附有一定量的习题与实践题供读者练习。

本书可以作为信息安全、信息与计算科学、计算机科学技术、通信工程等专业的本科生教材，也可以为从事信息安全、通信、电子工程等领域的技术人员提供参考。

图书在版编目(CIP)数据

密码编码与信息安全：C++ 实践/王静文，吴晓艺编著. --北京：清华大学出版社，2015

ISBN 978-7-302-39411-2

Ⅰ. ①密…　Ⅱ. ①王…　②吴…　Ⅲ. ①密码－加密技术　②信息安全－安全技术　Ⅳ. ①TN918.4　②TP309

中国版本图书馆 CIP 数据核字(2015)第 032188 号

责任编辑：刘　颖
封面设计：
责任校对：赵丽敏
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：22.25　　**字　　数**：540 千字
版　　次：2015 年 6 月第 1 版　　**印　　次**：2015 年 6 月第 1 次印刷
印　　数：1～　000
定　　价：　.00 元

产品编号：055283-01

前　言

1. 目的/目标

在密码编码与信息安全的教学与学习过程中,大多以理论教学为主,缺少实践教学,从而对所学的理论难以有较为深入的理解。本书重在以实践帮助读者理解密码编码与信息安全的基本原理。

本书主要讨论密码编码学与信息安全的基本原理,并以基本原理为基础,重在探讨C++的实现方法。通过逐步引导的方法,分析密码编码和信息安全的功能,并针对相应的功能采用C++语言加以实现。帮助读者掌握和理解密码编码与信息安全的原理,并将理论与实践有机结合,为对密码编码和信息安全有兴趣的读者提供参考。

2. 预备知识要求

本书主要讲述密码编码与信息安全的基本原理与实现方法,读者需有基本的C++语言知识,并能够编写简单的应用程序。若有基本的密码编码与信息安全的知识,则更有利于掌握书中的内容。

3. 学习方法

本书是以实践为主,帮助理解密码编码与信息安全的基本原理,从原理出发来分析和完成具体的实现方法与过程是理想的学习方法,切忌复制或抄袭代码,理解后独立完成相关实践内容,不仅有助于理解密码编码的原理,更有助于将理论转化为实践。

在本书的撰写过程中,每一部分的原理都通过分析和实践来完成,同时,将复杂的问题尽可能简化到易于理解,便于实现。读者在学习过程中,可以参考分解问题的方法,提高解决问题的能力。同时在程序设计过程中最好对各项功能进行单独测试,避免在总体完成后增大查找程序存在问题的复杂性。

4. 内容提要

本书共分为古典密码、现代对称密码、公钥密码算法、散列函数和数字签名五个部分,在古典密码中介绍了单表代替密码、移位密码、多表代替密码等古典密码算法。在现代对称密码中介绍了S-DES算法、DES算法、AES算法、IDEA算法、Blowfish算法和CAST-128等多种算法。在公钥密码算法中介绍了大数运算基本原理与实现方法、RSA算法、Diffie-Hellman算法、Elgamal算法等。在散列函数中介绍了MD4、MD5和SHA-1等算法。在数字签名中介绍了RSA数字签名方案、Elgamal数字签名方案和DSA数字签名方案等。并对书中出现的各种算法均给出了C++的具体实现。

5. 教学安排

本书作为教材可以根据教学学时安排具体内容，对于课时在60学时左右的教学计划可以选择大部分内容作为课程教学内容，对于课时在40学时左右的教学计划可以根据需要进行选择。例如：在第2部分内容中可以按照分类各选择一种算法作为教学内容，其中RC算法包含RC4、RC5和RC6算法，教学过程中则可以选择部分内容进行讲解，其余部分则可以作为教学参考。同样第4部分的散列算法也可以选择部分内容进行教学，算法细节不同，但基本结构有不少是相似的。

6. 错误

无论作者有多少发现错误的技巧，总有一些错误漏网，而读者往往最能发现错误，如果读者发现任何认为是错误的地方，请提出纠正建议，并发送电子邮件至 wang_jingwen@yeah.net，我们会非常感谢读者的帮助。

7. 编程环境

本书中的所有示例均采用G++编译器进行编译，编程环境为Code∷Blocks，保证了在Windows环境或Linux环境的兼容性，Code∷Blocks的下载网址为：http://www.codeblocks.org。若读者在Windows环境中使用VC进行处理，需做相应的修改以适应VC的编译器。

8. 致谢

本书在撰写过程中得到很多人的支持和帮助，特别要感谢何立国教授，对书中许多有关数学知识与相关证明的地方给出了很多宝贵的建议，使得本书更加完善。

编　者
2015年3月

目　录

第 2 部分　现代对称密码

第 3 部分 公钥密码算法

第5部分 数字签名

第 1 章 概　　述

随着网络与通信技术的发展，越来越多的信息通过通信网络、计算机以及电话、传真等技术手段被获取、存储和传输，在信息的处理过程中随时可能受到非法用户或非授权用户的访问、篡改或破坏，信息安全受到人们越来越多的关注和重视，密码编码则是信息安全的重要技术支撑。

1.1 密码学简介

密码学是一门研究信息保密的学科，密码学的基本任务就是通过一定的加密方法对信息提供保密性服务。

确保发送信息安全，同时接收者在获取信息后能正确获得原始信息是密码学的基本内容，这一过程由加密和解密两部分组成。加密通常是将明文进行编码，使其含义变得模糊或不易理解的过程，而解密是加密的逆过程，其作用是将密文变回其原始形式。在一个通信系统中，加密和解密的过程通常可以用图 1-1 所示的模型来表示。

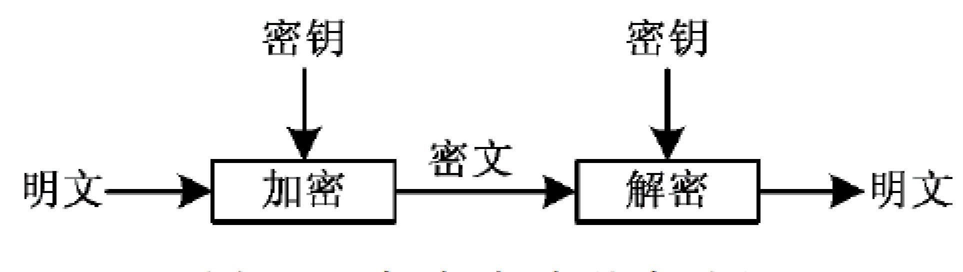


图 1-1　加密-解密基本过程

信息发送方利用加密密钥并采用一定的加密算法，对明文进行加密得到密文，然后将密文发送给接收方。接收方在收到密文之后，利用解密密钥和相应的解密算法将密文进行解密，从而获得原始的信息。在加密和解密过程中，密钥的使用方法又可以分为两类：一类是加密密钥与解密密钥相同，或者通过加密密钥很容易得到解密密钥，这类加密算法被称为单密钥系统；而另一类则是加密密钥和解密密钥不同，通过加密密钥无法或很难计算得到解密密钥，这类加密算法通常被称为双密钥系统。

对于单密钥系统，加密过程和解密过程可以表示为

$$\begin{aligned} C &= E_k(M), \\ M &= D_k(C)。 \end{aligned} \tag{1-1}$$

式中：M 表示明文，C 表示密文，E_k 表示加密密钥，D_k 表示解密密钥，且有 $D_k(E_k(M))=M$。式(1-1)也可以写成

$$C = E(K, M),$$
$$M = D(K, C)。$$

该式表示加密依赖于明文和密钥，解密依赖于密文和密钥，且加密密钥与解密密钥相同。对于双密钥系统，加密过程和解密过程可以表示为

$$C = E_{k1}(M),$$
$$M = D_{k2}(C), \tag{1-2}$$

且有 $D_{k2}(E_{k1}(M))=M$，式(1-2)也可以写成

$$C = E(K_1, M),$$
$$M = D(K_2, C)。$$

该式表示加密依赖于明文和加密密钥，解密依赖于密文和解密密钥，且加密密钥与解密密钥不同。

对称密码算法通常是单密钥系统，公钥密码算法通常是双密钥系统。

在加密和解密过程中包含以下 5 个基本要素：

(1) 明文——待加密的消息，可以是文本、图片、数字化的语音或数字化的视频等。

(2) 加密算法——是指对明文进行处理的方法或规则，使明文成为“不可理解”的密文。

(3) 密钥——是一种参数，它是在明文转换为密文或将密文转换为明文的算法中输入的数据。

(4) 密文——对明文进行加密后的输出，是不可理解的打乱的信息，通常可以通过算法还原。

(5) 解密算法——解密算法是指对密文进行解密的方法和规则，使密文还原为明文。加密算法和解密算法一般依赖于密钥。

目前所使用的加密算法和解密算法都属于基于密钥的算法，基于密钥的加密算法又可以分为对称密码算法和公开密钥算法。

对称密码算法通常是指加密密钥和解密密钥相同，或者通过加密密钥可以推算出解密密钥的算法，即单密钥系统。对称密码算法又称为传统密码算法，目前，商业上比较有影响力的传统加密算法有 DES(数据加密标准)算法、AES(高级加密标准)算法等。对称密码算法的通信模型如图 1-2 所示。

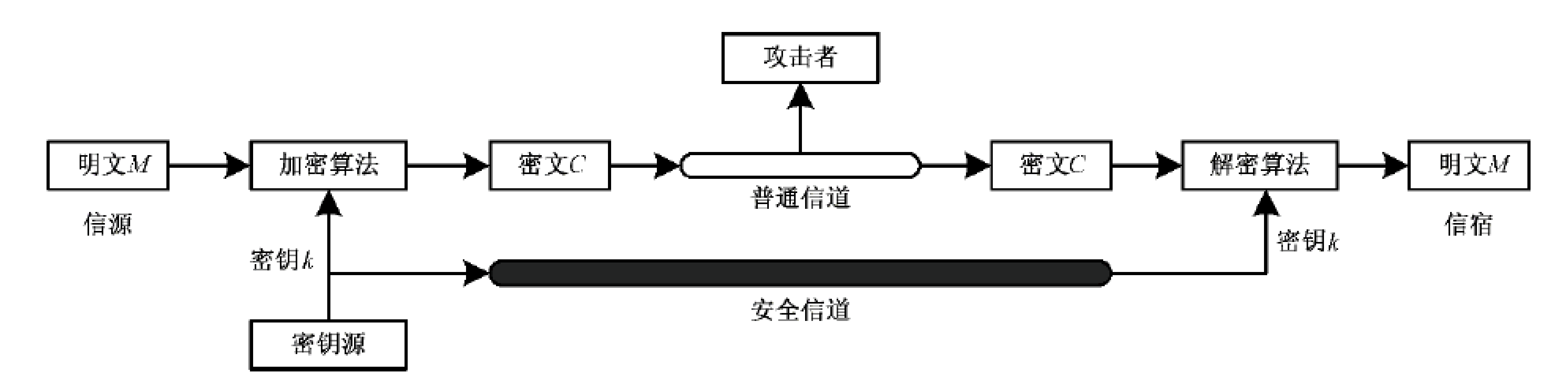


图 1-2 对称密码算法通信模型

在使用对称密码算法进行通信的过程中，密钥的传递需要“安全”地进行，若密钥使用普通信道进行传递则很容易受到攻击，当攻击者获得相应密钥之后，“密文”的保密性便不复存在。

公开密钥算法又称为公钥算法，公钥算法通常是指使用一个密钥进行加密，使用另一个密钥进行解密。在公钥算法中，加密密钥一般是公开的，通过加密密钥无法推算出解密密钥或很难推算出解密密钥，公钥系统为双密钥系统。

公钥加密算法也称为非对称密码算法，公钥加密算法不需要加密和解密双方互相信任。目前商业上比较流行的公钥加密算法有 RSA 算法、ECC（椭圆曲线）算法等。公钥密码算法的通信模型如图 1-3 所示。

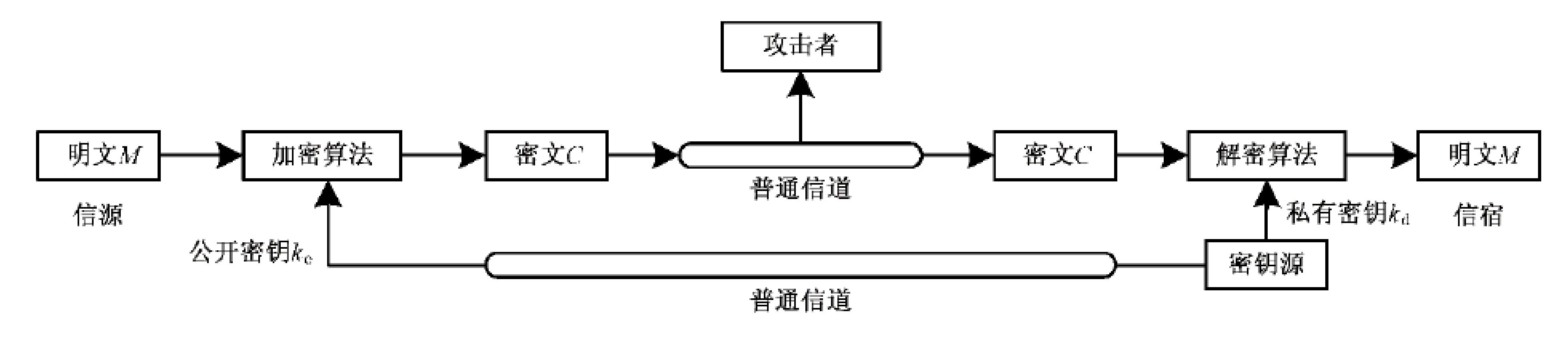


图 1-3　公钥密码算法通信模型

在公钥加密算法中，加密密钥的传递并不需要在一个特殊的安全信道中进行传递，即便攻击者获得加密密钥也无法通过加密密钥计算获得解密密钥，加密密钥可以“公开”，或者在普通的信道内传递加密密钥。

基于明文的加密方法可以分为流加密方法和分组加密方法。

流加密方法通常是对明文每次加密一个字符（或一位数据），这种加密方法主要在手工加密时代、机械加密时代使用或在语音通信中使用流加密算法，流加密算法具有的最大特点是具有较高的加密速度。

分组加密方法是指将明文分割成一定长度的组，然后使用同一个密钥和算法对每一组明文进行加密，输出的是相应长度的密文。目前使用的对称密码算法通常都是采用分组加密的方法进行加密和解密。

研究对信息进行编码的技术的学科称为密码编码学，其目的是实现对信息的隐蔽和保护。与密码编码学相对应，研究在不知道信息解密方法的情况下进行破解的科学称为密码分析学，有时候也称为破解密码。密码编码学和密码分析学通常是两个对立的学科。

1.2 信息安全遇到的威胁

密码编码与信息安全涉及的范围十分广泛，无论在军事、经济还是在人们的日常生活中，都在不同的程度上与密码编码和信息安全存在一定的联系。

计算机处理的信息主要分为两部分，一部分是计算机内部存储和处理的信息，另一部分是计算机之间互相交换的信息。计算机内部的信息不希望被非法人员访问，可以通过访问权限来限制非法用户读取计算机内部的信息。而计算机之间的信息传递中，传送者和接收者希望能保证传送信息的机密性和完整性。

计算机之间信息传递的安全是信息安全研究的主要内容，而密码编码学则是信息安全的基础。计算机网络的迅速发展也使得信息安全的问题日益突出，信息安全遇到了前所未有的威胁，在信息安全中，威胁通常是指侵犯安全的可能性，即利用安全系统的弱点和潜在危险，在破坏安全或引起危害的环境、行为或事件的情况下，会出现这些威胁。

信息安全中的安全攻击是指任何危害信息安全的活动，对信息的攻击的主要目的是从密文获得明文，更彻底的攻击是获得密钥。信息安全所遇到的威胁包含主动攻击和被动攻击两方面的威胁。

主动攻击是指以各种方式有选择性地进行信息破坏，例如：修改、删除、冒充和传播病毒等。主动攻击具有智能性、隐蔽性、多样性和破坏性的特点。

被动攻击是指在不干扰信息系统正常运行的情况下对计算机内部的信息或计算机通信中的信息进行截取、破译和分析。

主动攻击难以防止，但容易检测，因此，针对主动攻击其重点在于检测并从破坏中得到恢复。被动攻击与主动攻击相反，被动攻击可以防止但难以检测，因此，针对被动攻击其重点在于如何预防。

1.3 密码编码和信息安全提供的服务

密码编码与信息安全主要保障信息的安全利用，提供的服务主要有以下几种：

（1）机密性

机密性主要是针对私有信息的保护，指一个实体（可以是个人或集体）把自己的数据存放到一个合适的位置，可以方便地读取或存放，而其他未经授权的实体则无法读取和破坏，或未授权用户无法理解信息的内容。

如果信息保存在未被保护的地方，或者信息在不安全的信道内进行传输都存在信息泄密的可能性，信息安全的一个重要任务就是保护信息的机密性，一般采用加密技术来实现机密性保护。

（2）数据完整性

数据完整性具体体现在对交换信息的保护，指若干个存在利益相关的实体（可以是个人也可以是集体）合作完成相关的交易（或事情），在交易过程中不被不相关的实体干扰、窃听和破坏。同时又保证各交易方不能进行欺骗或至少不能摆脱对自己行为的责任。

同机密性保护类似，完整性服务的要求同样出现在存储或传输过程中，这时，信息存在被篡改的可能性，在这种情况下，信息安全的任务就是保护信息的完整性。在密码学中可以采用数据加密、报文鉴别和数字签名等技术来实现数据完整性保护。

（3）鉴别

鉴别通常用于权利保护，是指身份鉴别有关的保护。权利保护的权利是指一个实体利益的知识产权，或代表一个实体责任的控制权限，例如：管理员、负责人的权限被非法盗用，软件、电影以及其他数字产品被盗版。权利保护则使非法人员难以盗用他人权限和盗版，或在盗用后无法去除痕迹，使这些痕迹具有可追踪性。鉴别通常可以通过数据加密、数字签名或鉴别协议等技术来实现。

（4）抗抵赖

抗抵赖是指用于阻止通信实体的抵赖行为及其相关内容的相关服务。人们为了自身或团体的利益可能抵赖曾经发送过的消息，因此，当消息发送后，接收方需要向他人证实该消息确实是从所声称的发送方发送。抗抵赖一般可以通过数字签名技术或协同可信机构、证书机构来完成相关服务。

密码编码学和信息安全的主要任务是从理论上和实践上来解决上述 4 个问题。

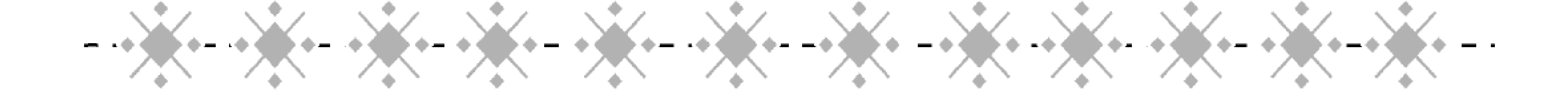
1.4 习题

1. 什么是对称密钥加密算法？
2. 什么是公开密钥加密算法？
3. 为什么对称密钥加密算法的密钥传递需要在安全的信道中进行？
4. 为什么公开密钥加密算法的密钥传递可以在普通的信道中进行？
5. 什么是主动攻击？
6. 什么是被动攻击？
7. 什么是信息安全中的机密性服务？
8. 什么是信息安全中的完整性服务？
9. 什么是信息安全中的鉴别服务？
10. 什么是信息安全中的抗抵赖服务？

第 1 部分

古 典 密 码

为了能够更好地理解密码编码体制，在本部分中，首先从古典密码编码技术着手来说明密码编码的基本技术，古典密码编码技术大多数采用的是手工或机械的方法对明文进行加密，对密文进行解密，在现代来说基本已无安全可言，但是理解古典的密码编码技术有助于理解密码编码的基本实现；同时，由于其编码方法相对简单，在计算机上比较容易实现，对后续加密方法与解密方法的学习有一定的帮助作用。



第 2 章 古典密码编码技术

2.1 单表代替密码

2.1.1 单表代替密码编码原理

单表代替密码是一种固定明文字符集到密文字符集的映射，是古代密码编码技术中的一种比较简单的编码技术，属于简单的代换密码，即一个字符用另一个字符进行代换，这种代换可以用映射表示为

$$f: S^L \to C^n 。 \tag{2-1}$$

假设有明文 $S=(s_1 s_2 s_3 \cdots)$，则相应的密文为

$$C = E_k(S) = f(s_1)f(s_2)f(s_3)\cdots 。 \tag{2-2}$$

若具有 n 个字符的明文字符集 $A=\{a_0, a_1, \cdots, a_{n-1}\}$，则相应的与明文字符集具有映射关系的密文字符集为 $A'=\{f(a_0), f(a_1), f(a_2), \cdots, f(a_{n-1})\}$。

单表代替密码加密的映射关系是一个一对一映射，映射 f 的逆映射为 f^{-1}，解密密钥就是一个固定的代换字母表，如果存在密文 $C=(c_1 c_2 \cdots)$，则解密过程为

$$S = D_k(C) = f^{-1}(c_1)f^{-1}(c_2)\cdots 。 \tag{2-3}$$

单表代替密码中比较典型的密码算法是凯撒（Caesar）密码，其本质就是构造不同字母的映射表，通过映射表来完成对数据的加密和解密。

2.1.2 单表代替密码算法实现

单表代替密码算法的实现过程主要包括两部分内容，一部分是加密过程，另一部分是解密过程，下面的实现过程就是针对历史上有名的凯撒密码来完成的。

加密过程是针对明文文件进行加密，采用读取字符的方法，对每一个字符通过映射的方法进行加密，完成加密后写入加密文件。在本节的示例中所包含的字符由 26 个小写字母和空格组成的，因过程比较简单，单表代替密码算法的实现直接采用函数的方法来实现，具体代码如下。

程序清单 2-1

```
#include<iostream>
#include<fstream>
#include<cstdlib>
```

```
using namespace std;
const char c[27]={'d','j','k','z','u','x','c','m','l','i','w','b','v','n',
                   'o','p','q','a','r','s','g','h','f','t','y','e',' '};
void encryption(ifstream& fin,ofstream& fout);
int main()
{
    ifstream fin;
    ofstream fout;
    fin.open("file1_1.in");
    if(fin.fail())
    {
        cout<<"File open error!(Input)"<<endl;
        exit(1);
    }
    fout.open("file1_1.out");
    if(fout.fail())
    {
        cout<<"File open error!(Outpot)"<<endl;
    }
    encryption(fin, fout);
    fin.close();
    fout.close();
    return 0;
}
void encryption(ifstream& fin,ofstream& fout)
{
    char next;
    char ch;
    int i;
    while(fin.get(next))
    {
        if(next>='a' && next<='z')
        {
            i=next-'a';
            ch=c[i];
            fout<<ch;
        }
        else
        {
            fout<<' ';
        }
    }
}
```

在程序中的输入文件为 file1_1.in，输出文件为 file1_1.out。

运行示例：

输入文件 file1_1.in 的内容为：we will attack tomorrow morning

输出文件 file1_1.out 的内容为：fu flbb dssdkw sovoaaof voanlnc

程序说明：

1. 26 个字母与空格的单表替换是通过 const char c[27]来完成的，具体的明文与密文之间字符的映射关系根据具体情况来确定，读者在实践过程中可以自己定义明文与密文之间的映射关系。

2. 替换函数 void encryption(ifstream& fin,ofstream& fout)的参数是文件输入流和文件输出流，输入和输出文件在主函数中定义。

3. 在读取文件数据的时候采用的方法是 fin.get(next)，而不是 fin>>next，当采用 fin>>next 来读取数据的时候，程序将会忽略明文文件中的空格。

解密过程是加密的逆映射，在解密过程中可以查找密文所在的索引来确定明文的内容，在本例中的密文与明文的对应关系可以通过图 2-1 来确定。

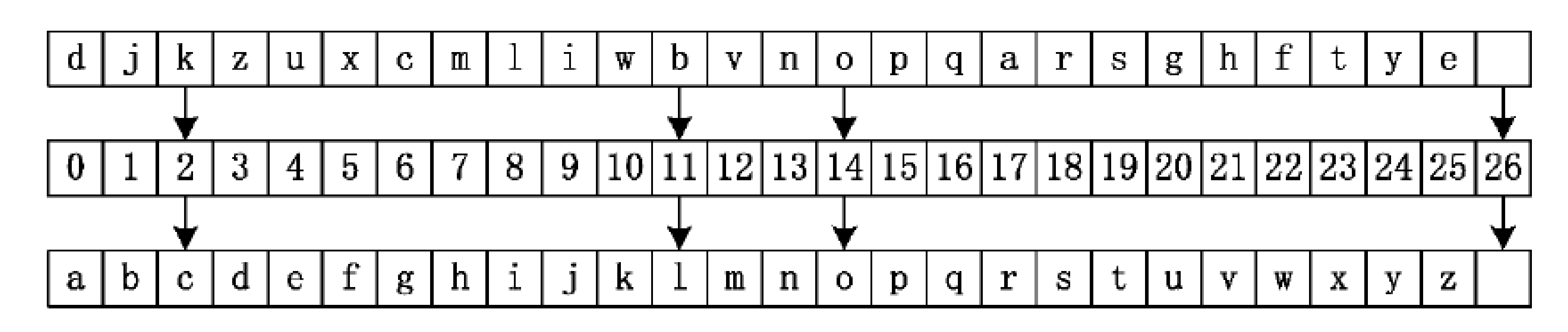


图 2-1 密文与明文对应关系

假设密文字母为 k，那么它在字符数组中的索引为 2，与明文相对应的字母为 c，假设密文字母为 b，那么它在字符数组中的索引为 11，与明文相对应的字母为 l。通过上述过程就可以得到相应的密文-明文对应关系，程序的实现可以仿照加密过程的方法来实现，解密过程的函数代码如下。

程序清单 2-2

```
void decryption(ifstream& fin,ofstream& fout)
{
    char ch;
    char chout;
    while(fin.get(ch))
    {
        for(int i=0;i<=26;i++)
        {
            if(ch==c[i])
            {
                if(i==26)
                {
                    fout<<' ';
                }
                else
                {
```

```
                        chout=char('a'+i);
                        fout<< chout;
                    }
                }
            }
        }
}
```

函数参数 ifstream& fin 是要读取的加密了的文件,参数 ofstream& fout 是解密后要输出的文件,程序中读取文件中的内容同样采用 fin.get(ch),其目的与加密过程读取文件时一样,防止漏读空格字符,字符转换通过 char('a'+i)完成,其他部分和加密过程的程序类似,可以参考加密过程的程序完成。

2.2 移位密码

2.2.1 移位密码算法原理

移位密码又称为移位代换密码,是单表代换密码的一种,它的加解密过程可以用以下方式表示:

$$\begin{aligned} c &= E_k(s) = (s+k) \bmod n, \\ s &= D_k(c) = (c-k) \bmod n, \end{aligned} \tag{2-4}$$

其中,c 表示密文字符,s 表示明文字符,k 表示移位的数字,n 表示代换字符集的字符总个数,当字符集为 26 个字母时的移位算法就是凯撒密码。

2.2.2 移位密码算法实现

移位算法的 C++ 实现比较简单,可以通过构造一个简单的类来实现,类中包含一个加密函数和一个解密函数,加密和解密过程都是针对文件来进行,类的声明如下。

程序清单 2-3

```
class Shift
{
public:
    Shift();
    void encryption(ifstream& fin, ofstream& fout, int n);
    void decryption(ifstream& fin, ofstream& fout, int n);
};
```

其中 encryption 函数用于对文件进行加密,函数的参数为输入文件、输出文件和移位的量,输入文件为需加密的原文,输出文件为加密后的密文。decryption 函数用于对密文文件进行解密,函数参数的输入文件为密文文件,输出文件为解密后的明文。

加密函数 encryption 的实现过程如下。

程序清单 2-4

```
void Shift::encryption(ifstream&fin, ofstream& fout, int n)
```

```
{
    char next;
    while(fin.get(next))
    {
        fout<< char((int(next)+ n)%128);
    }
}
```

解密函数 decryption 的实现过程如下。

程序清单 2-5

```
void Shift::decryption(ifstream& fin, ofstream& fout, int n)
{
    char next;
    while(fin.get(next))
    {
        fout<< char((int(next)- n)%128);
    }
}
```

上述加密方法和解密方法都是针对整个 ASCII 编码表进行的，因此在实现过程中使用 char((int(next)＋n)％128)来进行加密运算，使用 char((int(next)－n)％128)来进行解密运算，在实现过程中同样需要注意读取文件的方法，在读取文件时采用 fin. get(next)来读取文件中的字符，防止漏读空格字符。

2.3 乘数密码

2.3.1 乘数密码算法原理

乘数密码算法的基本原理类似于单表替换密码算法或移位密码算法，加密函数和解密函数可以用下式表示：

$$\begin{aligned} c &= E_k(s) = (ks)\bmod n, \\ s &= D_k(c) = (k^{-1}c)\bmod n, \end{aligned} \tag{2-5}$$

式中：n 表示明文字符集中字符的总个数，s 表示明文字符，c 表示密文字符，k 表示密钥。

乘数密码算法需满足两个条件：

1. $0\leqslant k<n$；

2. $\gcd(k,n)=1$。

条件 2 表示 k 与 n 互素，若 k 与 n 不满足互素条件，则明文字符集与密文字符集不能构成一一映射。

图 2-2 是不同密钥的明密文对示意图，在 $k=9$ 时，9 与 26 互素，得到的明密文对为一一映射，$k=4$ 时，4 与 26 非互素，有公因子 2，得到的明密文对非一一映射。例如：明文的 b 和 o 所对应的密文都为 e，明文的 a 和 n 对应的密文都为 a。

乘数密码算法在得到密钥之后，计算得到明密文对，实际上就形成了单表代替密码，因

明文	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
k=9	a	j	s	b	k	t	c	l	u	d	m	v	e	n	w	f	o	x	g	p	y	h	q	z	i	r
k=4	a	e	i	m	q	u	y	c	g	k	o	s	w	a	e	i	m	q	u	y	c	g	k	o	s	w

图 2-2 不同密钥明密文对示意图

此其解密的过程也可以完全参考单表代替密码来实现，若使用公式(2-5)中的解密算法，需要计算 k 的模 n 的乘法逆元，模 n 的乘法逆元的计算方法见下一节扩展的欧几里得算法。

2.3.2 扩展的欧几里得算法

扩展的欧几里得算法是以欧几里得(Euclid)算法为基础发展而来的，它的功能是计算两个正整数的最大公因子。

假设 a 和 b 是两个正整数，如果正整数 c 满足：

(1) c 是 a 和 b 的因子；

(2) a 和 b 的任何因子都是 c 的因子；

此时，c 称为 a 和 b 的最大公因子，用 $\gcd(a,b)$表示。

定理 对于任意非负整数 a 和 b，有

$$\gcd(a,b) = \gcd(b,a \bmod b)。\tag{2-6}$$

欧几里得算法就是通过重复使用上述定理来计算最大公因子的，其计算过程用伪码可以表示如下：

```
Euclid(a,b)
{
    if(b==0)
    {
        return a;
    }
    else
    {
        return Euclid(b,a mod b)
    }
}
```

在乘数密码的解密过程中，即可以采用单表代替密码的方法进行解密，也可以先计算密钥 k 的模 n 的乘法逆元 k^{-1}，然后再利用解密公式进行解密，计算乘法逆元可以采用扩展的欧几里得算法。

定理 对于任意的正整数 a 和 b，存在整数 s 和 t，使得

$$as + bt = \gcd(a,b)。\tag{2-7}$$

假设 $\gcd(a,b)=1$(a 和 b 互素)，此时存在一个 a 模 b 的乘法逆元 a^{-1}，即

$$aa^{-1} \equiv 1(\bmod\ b)。\tag{2-8}$$

由于 $\gcd(a,b)=1$，由上述定理，我们知道存在 s 和 t，使得 $as+bt=1$，该式可以转换为 $as=1-bt$，于是

$$as \equiv 1 \pmod b。\tag{2-9}$$

所以 s 为 a 模 b 的乘法逆元。

要解决计算 s 和 t 的问题，目前比较常用的计算方法是扩展的欧几里得算法，扩展的欧几里得算法实际上就是欧几里得算法的变形，扩展的欧几里得算法的伪码如下。

```
ExtEuclid(a,b)
{
    if(b==0)
    {
        return(a,1,0);
    }
    (d1,s1,t1)=ExtEuclid(b, a%b);
    d=d1;
    s=t1;
    t=s1-(a/b)*t1;
    return (d,s,t);
}
```

示例 2-1　假设 $a=99, b=78$，计算满足 $as+bt=\gcd(a,b)$ 的 s 和 t。

解　计算过程采用 ExtEuclid()，递归调用 ExtEuclid(99,78)得到结果如下：

	a	b	a/b	d	s	t
1	99	78	1			
2						
3						
4						
5						
6						

由于 78≠0，因此，再次调用 ExtEuclid()，此时函数的参数为 78 和 21，因为 99%78=21，再次计算结果如下：

	a	b	a/b	d	s	t
1	99	78	1			
2	78	21	3			
3						
4						
5						
6						

重复递归调用的过程，一直到 b=0 结束，此时返回的值为(3,1,0)，整个递归调用的计算结果如下：

	a	b	a/b	d	s	t
1	99	78	1			
2	78	21	3			
3	21	15	1			
4	15	6	2			
5	6	3	2			
6	3	0	—	3	1	0

上述计算结果的第 5 行的 d,s,t 的计算来源于第 6 行的结果,在第 6 行中有 d1＝3,s1＝1 和 t1＝0,在第 5 行的计算中有 d＝d1＝3,s＝t1＝0 和 t＝s1－(a/b) * t1＝1－2 * 0＝1,得到的计算结果如下:

	a	b	a/b	d	s	t
1	99	78	1			
2	78	21	3			
3	21	15	1			
4	15	6	2			
5	6	3	2	3	0	1
6	3	0	—	3	1	0

重复上述计算过程,一直到整个过程计算完毕,其计算结果如下:

	a	b	a/b	d	s	t
1	99	78	1	3	－11	14
2	78	21	3	3	3	－11
3	21	15	1	3	－2	3
4	15	6	2	3	1	－2
5	6	3	2	3	0	1
6	3	0	—	3	1	0

最终计算结果为:$s=-11,t=14$。因为 $99\times(-11)+78\times14=3$,因此得到的 s 和 t 满足 $as+bt=\gcd(a,b)$。

示例 2-2 计算 60 模 13 的乘法逆元。

解 由于 gcd(60,13)＝1,所以存在乘法逆元,计算过程与示例 2-1 相同,计算结果如下:

	a	b	a/b	d	s	t
1	60	13	4	1	5	－23
2	13	8	1	1	－3	5
3	8	5	1	1	2	－3
4	5	3	1	1	－1	2
5	3	2	1	1	1	－1
6	2	1	2	1	0	1
7	1	0	—	1	1	0

由于 $60\times5+13\times(-23)=300-299=1$，因此可以得到 5 是 60 模 13 的乘法逆元，因为 $(60\times5)\%13=300\%13=1$。

2.3.3 乘数密码算法实现

乘数密码算法的实现过程中首先要找到候选密钥，然后在候选密钥中选择相应的密钥，在程序实现的过程中可以通过两个函数来实现，一个用于计算最大公约数，另一个是将所有满足 $\gcd(k,n)=1$ 的 k 输出。

计算最大公约数的函数如下。

程序清单 2-6

```
int gcd(int n,int k)
{
        if(n==0)
      {
          return k;
      }
      if(k==0)
      {
          return n;
      }
      return gcd(k, n%k);
}
```

函数采用了易于理解的递归算法，其返回值为最大公约数。

输出满足 $\gcd(k,n)=1$ 的所有 k 的函数代码如下。

程序清单 2-7

```
void getGcd(int n)
{
      int i;
      vector<int> v;
      for(i=1;i<n;i++)
      {
          if(gcd(n,i)==1)
          {
              v.push_back(i);
          }
      }
      for(i=0;i<v.size();i++)
      {
          cout<<v[i]<<" ";
      }
      cout<<endl;
}
```

该函数中采用 vector 来临时存储获得的满足 $\gcd(k,n)=1$ 的 k，在编程时需包含相应的头文件，在计算完所有的 k 之后，再将 k 输出，供加密过程选择。

在计算得到密钥 k，并选择完相应的密钥之后，乘数密码算法的加密过程和解密过程可以通过两种方法来进行，一种是以单表代替密码的方法来进行，一种是以乘数密码算法的加密和解密方法进行，以单表代替密码的解密方法进行解密。加密过程可以参照程序清单 2-1 来完成，解密过程可以参照程序清单 2-2 来完成。

2.3.4 扩展的欧几里得算法的实现

在 2.3.2 节中描述了扩展的欧几里得算法，并给出了相应的例子，说明了扩展的欧几里得算法的基本原理。由于该算法在后续的公钥密码算法中有较大的用途，因此本节给出算法实现的完整代码以及解释，方便读者理解扩展的欧几里得算法，完整的程序代码如下。

程序清单 2-8

```
#include<iostream>
using namespace std;
struct ExtEuc
{
    int d;
    int s;
    int t;
};
ExtEuc ExtEuclid(int,int);          //扩展的欧几里得算法函数声明
int main()
{
    ExtEuc eu;
    int a,b;
    cout<<"Please input two integers: ";
    cin>>a>>b;
    eu=ExtEuclid(a,b);
    cout<<"d="<<eu.d<<" "<<"s="<<eu.s<<" "<<"t="<<eu.t<<endl;
    return 0;
}
ExtEuc ExtEuclid(int a,int b)
{
    ExtEuc eu,eu1;
    if(b==0)
    {
        eu1.d=a;
        eu1.s=1;
        eu1.t=0;
        return eu1;
    }
    eu1=ExtEuclid(b,a%b);
    eu.d=eu1.d;
```

```
32        eu.s=eu1.t;
33        eu.t=eu1.s-(a/b)*eu1.t;
34        return eu;
35  }
```

由于整个程序比较简单,程序仅以函数的形式来完成,第 3 行到第 8 行定义了要处理的数据的结构体,将 d,s,t 封装在结构体中,方便进行数据处理,其中 d 表示 a 和 b 的最大公约数,当 d=1 时,s 为 a 模 b 的乘法逆元,t 为 b 模 a 的乘法逆元。第 20 行到第 35 行的函数 ExtEuclid(int,int)实现扩展了欧几里得算法,函数参数为整型数据 a 和 b,其返回值类型为 ExtEuc。函数的计算方法采用递归,主函数实现数据的输入和计算结果的输出。

2.4 多表代替密码

在上述加密方法中,密文字母和明文字母之间是一对一映射,其加密方法均无法掩盖明文数据的统计特性,通过统计计算密文字母的统计特性,比较容易找出明文和密文之间的对应关系,从而达到破解密文的目的。

以英文文章为例,不同的字母在文章中出现的频率是不同的,美国密码学家 William Freidman 在对大量英文文章进行统计后,得到不同英文字母的普遍使用频率,不同英文字母(不区分大小写)的使用频率见表 2-1。

表 2-1 英文字母使用频率

字母	频率	字母	频率	字母	频率
A	0.0856	J	0.0133	S	0.0607
B	0.0139	K	0.0042	T	0.1045
C	0.0279	L	0.0339	U	0.0249
D	0.0378	M	0.0249	V	0.0092
E	0.1304	N	0.0707	W	0.0149
F	0.0289	O	0.0797	X	0.0017
G	0.0199	P	0.0199	Y	0.0199
H	0.0528	Q	0.0012	Z	0.0008
I	0.0627	R	0.0677		

根据英文字母的使用频率,很容易推测出密文中的不同字符与明文中的不同字符的对应关系,从而破解密文。

以同样的方式还可以统计双字母和三字母的统计特性,例如:在双字母中 TH、HE、ER 和 AN 等是使用频率相对较高的双字母组合,而在三字母中则 THE、ING、AND 和 HER 等属于使用频率较高的三字母组合。利用双字母和三字母的统计特性,可以更有效地提高简单密码算法的准确性。

对单表代替密码进行改进,产生多表代替密码,多表代替密码隐藏了明文的统计特性,明文的结构信息被隐蔽了。

多表代替密码有两个共同的特征,其一是采用相关单表代换规则,其二是密钥决定代换

的具体方法。

在这类密码算法中，比较典型的多表代替密码是维吉尼亚(Vigenere)密码和希尔(Hill)密码，希尔密码实际上也是多表代替密码的另一种变体。

2.4.1 维吉尼亚密码原理

维吉尼亚密码是多表代替密码中最简单的一种，它的代换规则和凯撒密码类似，每个代换表是明文字母移位 0 到 25 次后得到的代换单表，其代换表如表 2-2 所示。

表 2-2 维吉尼亚密码编码表

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
b	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
c	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
d	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
e	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
f	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
g	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
h	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
i	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
j	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
k	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
l	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
m	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
n	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
o	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
p	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
r	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
s	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
t	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
u	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
v	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
w	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
x	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

在表 2-2 中，第一行表示明文字母，第一列表示密钥字母，中间的大写字母表示加密后的密文，维吉尼亚密码的加密过程为：首先确定密钥，密钥在后续的加密过程中循环使用，然后根据密钥和明文来确定密文。

现设密钥为 cipher，明文为 seeyoulate，其加密结果如下：

密钥：cipherciph

明文：seeyoulate

密文：UMTFSLNIIL

明文的第 1 个字母为 s，密钥的第 1 个字母为 c，密文则是第 c 行第 s 列对应的字母 U，在明文中第二个字母和第三个字母都是 e，但由于密钥不同，因此加密得到的密文也不同，这样，密文隐藏了明文的统计特性。由于密钥的长度为 6，那么在第 6 个字母之后密钥循环使用，即第 7 个明文字母加密时使用的是密钥的第 1 个字母，以此类推。因此，与单表代替密码相比维吉尼亚密码具有更高的安全性。

2.4.2 维吉尼亚密码实现

本节的示例以完整的面向对象方法来实现，通过构造 vigenere 类对所需要处理的数据进行封装，同样，相应的函数也封装在类中。

完整的实现共包括三个相应的文件，vigenere. h、vigenere. cpp 和 main. cpp，在 vigenere. h 中主要完成对类的声明，vigenere. cpp 文件主要完成 vigenere. h 中声明函数的定义，main. cpp 中主要是对象的调用。

vigenere. h 的代码见程序清单 2-9。

程序清单 2-9

```
#ifndef VIGENERE_H
#define VIGENERE_H
#include< string>
using namespace std;
class Vigenere
{
    private:
        int cipherTable[26][26];        //数字编码表
        string key;                     //密钥
        string explicitly;              //待处理的明文
        string cipherText;              //加密后的密文
        string explicitlyText;          //解密后的明文
    public:
        Vigenere();
        void init();                    //初始化 cipherTable
        void setKey();                  //设置加密密钥
        void setExplicitly();           //设置明文
        void encryption();              //加密明文，获得密文
        void decryption();              //将密文转换为明文
        void outPut();                  //将处理结果输出
        int getPosition(int p,int n);   //获取密文解密的位置
};
#endif
```

类中各成员变量和成员函数的功能在注释中给出，其中 cipherTable[26][26]构造了一个特殊的表，其功能是将 Vigenere 表中的字母全部由数字替换，目的是简化程序并提高效率，具体实现的方法在程序定义中说明，其他函数的实现方法也在后面加以说明。

类中 init()函数代码见程序清单 2-10。

程序清单 2-10

```
void Vigenere::init()
{
    int i,j;
    for(i=0;i<26;i++)
    {
        for(j=0;j<26;j++)
        {
            cipherTable[i][j]=(i+j)%26;
        }
    }
}
```

程序中 cipherTable 的作用是将维吉尼亚密码编码表数字化，将原来用字符表示的编码表改为用数字表示的编码表，编码时根据明文字符和密钥字符所在的列和行将密文转换为数字形式来表示，例如明文字符为“c”，则表示字符所在的列为 int(c－'a')，结果为第 2 列，密钥字符为“d”，密钥字符所在的行为 int(d－'a')，结果为第 3 行，则对应密文的数字为(2＋3)％26，而对应的密文则为 char(int('A')＋(2＋3)％26)，得到密文为“F”，这个过程将字符进行数字化处理，简化了加密和解密的过程。

init()函数在构造函数内直接调用，在构造 Vigenere 类的对象时就对密码编码表进行初始化操作。

类中的 setExplicitly()函数的作用是设置明文，void setKey()的作用是设置密钥，在本例中通过控制台来输入明文和密钥，也可以通过文件或其他方法来输入明文和密钥，函数 encryption()的作用是将明文加密获得密文，其实现代码见程序清单 2-11。

程序清单 2-11

```
01  void Vigenere::encryption()
02  {
03      cipherText=explicitly;          //初始化密文为明文
04      unsigned int i;
05      int m,n;                        //计算密文所在的行列参数用
06      for(i=0;i<explicitly.length();i++)
07      {
08          m=int(explicitly[i]-'a');
09          n=int(key[i%(key.length())]-'a');
10          cipherText[i]=char('A'+cipherTable[n][m]);
11      }
12  }
```

加密过程的基本方法是：逐个字符读取明文和密钥，并通过明文字符和密钥字符计算出明文字符所在的列和密钥字符所在的行，然后再计算得到密文字符，这三步分别由程序清单 2-11 中的第 8 行、第 9 行和第 10 行中的代码完成。

解密过程由函数 decryption()来完成,完整代码见程序清单 2-12。

程序清单 2-12

```
01  void Vigenere::decryption()
02  {
03      explicitlyText=cipherText;          //初始化解密明文为密文
04      unsigned int i;
05      int m,n;
06      for(i=0;i<cipherText.length();i++)
07      {
08          n=int(key[i%(key.length())]-'a');
09          m=getPosition(i,n);
10          explicitlyText[i]=char('a'+m);
11      }
12  }
```

解密过程的基本方法是:首先获得密钥字符所在的行,然后通过密钥字符所在的行和密文字符来获得明文字符所在的列,最后计算获得明文字符。程序中第 8 行的作用是确定密钥字符所在的行,通过 i%(key.length())来确定使用第几个密钥,通过 key[i%(key.length())]确定密钥的字符,然后通过 key[i%(key.length())]-'a'确定密钥在表中的第几行。

计算明文字符所在的列通过函数 getPosition()来实现,函数参数为密文所在位置和密钥所在行,函数的返回值为明文所在的列,具体实现代码见程序清单 2-13。

程序清单 2-13

```
01  int Vigenere::getPosition(int p,int n)
02  {
03      int position;
04      int i;
05      for(i=0;i<26;i++)
06      {
07          if(char(cipherText[p])==char('A'+cipherTable[n][i]))
08          {
09              position=i;
10              break;
11          }
12      }
13      return position;
14  }
```

getPosition()函数通过当前密钥所在的行来获得该行的密文表,然后通过密文字符与密文表的字符比较来获得明文的位置,这是一个比较简单并易于实现的方法。代码行的第 5 行到第 12 行,通过对比的方法在表中确定明文所在的列。

在类的成员函数中还有一个输出函数,其功能是输出明文、密文和解密后的明文,函数比较简单,读者可以自行完成。

2.4.3 希尔密码的原理

希尔(Hill)密码是1929年由数学家 Lester Hill 发明的，加密算法将 m 个连续的明文字母替换成 m 个密文字母，并且由 m 个线性等式决定变换的方式。在线性等式里字母与数值一对一映射，a→0，b→1，…，z→25，例如当 $m=3$ 时系统可以表示为

$$\begin{aligned} c_1 &= (k_{11}p_1 + k_{12}p_2 + k_{13}p_3) \bmod 26, \\ c_2 &= (k_{21}p_1 + k_{22}p_2 + k_{23}p_3) \bmod 26, \\ c_3 &= (k_{31}p_1 + k_{32}p_2 + k_{33}p_3) \bmod 26。 \end{aligned} \tag{2-10}$$

用列矢量和矩阵表示为

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \bmod 26。 \tag{2-11}$$

将式(2-11)用通式可以表达如下：

$$\boldsymbol{c} = \boldsymbol{K}\boldsymbol{p} \bmod 26 \tag{2-12}$$

式中：$\boldsymbol{c}$ 和 $\boldsymbol{p}$ 是长度为 m 的列矢量，分别代表密文和明文，$\boldsymbol{K}$ 是一个 $m\times m$ 矩阵，代表加密密钥，运算按模 26 执行。

解密过程是加密过程的逆过程，希尔密码算法的解密过程可以表示为

$$\boldsymbol{p} = \boldsymbol{K}^{-1}\boldsymbol{c} \bmod 26 \tag{2-13}$$

式中：$\boldsymbol{K}^{-1}$是加密密钥 $\boldsymbol{K}$ 的逆。

希尔密码算法在实际计算中字母与数字对应的多少是根据实际情况来确定的。在很多应用算法中，加入了“.”、“?”和空格“␣”，使得解密后得到的明文可以与原来的明文一致，其映射关系见表 2-3。

表 2-3 29 个字母的数字-字母映射表

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	.	?	␣
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

采用 29 个字母映射的加密方法与采用 26 个字母映射的计算方法是完全一致的。

希尔密码算法中涉及在模运算下计算矩阵的逆，因此，加密矩阵必须在模运算下可逆才能用于加密计算。

希尔密码算法采用的加密和解密矩阵通常为 2×2 矩阵，解密矩阵通过计算加密矩阵在模运算下的逆获得。

如果加密矩阵为

$$\boldsymbol{K} = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

可逆，即 $ad-bc\neq 0$，则其逆矩阵为

$$\boldsymbol{K}^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}。 \tag{2-14}$$

在计算加密矩阵在模运算下的逆时，必须存在 $ad-bc$ 的乘法逆元。如果采用 26 个字

母的映射，那么，$ad-bc$ 模 26 必须是 1，3，5，7，9，11，15，17，19，21，23 或 25 中的一个；否则，不能进行解密。

示例 2-3 采用映射表为 26 个字母与数字的映射，加密矩阵为 $\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 9 & 4 \\ 5 & 7 \end{bmatrix}$，试计算解密矩阵。

解 $ad-bc=9\times7-4\times5=63-20=43\equiv17 \bmod 26$。

由于 17 存在模 26 的乘法逆元，因此，加密矩阵的逆为

$$\begin{bmatrix} \frac{7}{17} & \frac{-4}{17} \\ \frac{-5}{17} & \frac{9}{17} \end{bmatrix} \bmod 26。$$

计算得 17 模 26 的乘法逆元为 23，即 $17\times23=1 \bmod 26$，于是有

$$\begin{bmatrix} \frac{7}{17} & \frac{-4}{17} \\ \frac{-5}{17} & \frac{9}{17} \end{bmatrix} \bmod 26 \equiv \begin{bmatrix} 7\times23 & -4\times23 \\ -5\times23 & 9\times23 \end{bmatrix} \bmod 26$$

$$\equiv \begin{bmatrix} 161 & -92 \\ -115 & 207 \end{bmatrix} \bmod 26$$

$$\equiv \begin{bmatrix} 5 & 12 \\ 15 & 25 \end{bmatrix} \bmod 26。$$

对计算结果进行验算，有

$$\begin{bmatrix} 9 & 4 \\ 5 & 7 \end{bmatrix} \begin{bmatrix} 5 & 12 \\ 15 & 25 \end{bmatrix} \bmod 26 \equiv \begin{bmatrix} 45+60 & 108+100 \\ 25+105 & 60+175 \end{bmatrix} \bmod 26$$

$$\equiv \begin{bmatrix} 105 & 208 \\ 130 & 235 \end{bmatrix} \bmod 26$$

$$\equiv \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \bmod 26。$$

乘法逆元的计算方法见 2.3.2 节的扩展的欧几里得算法。

在程序设计过程中要注意除法问题，当存在除法时需要先计算分母的乘法逆元，将除法转换为乘法运算。否则，得不到正确的运算结果。

希尔密码算法在一般的情况下是采用二阶方阵作为加密密钥。采用二阶密钥隐蔽了明文的二元统计信息，若需要隐蔽二元以上的明文统计信息，则需要采用更高阶的方阵作为密钥。例如，有时候会采用三阶方阵作为加密密钥。

三阶以上的矩阵计算逆矩阵可以采用初等变换法或伴随矩阵法等。

示例 2-4 采用映射表为 26 个字母与数字的映射，加密矩阵为 $\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 5 & 6 & 0 \end{bmatrix}$，试采用初等变换方法计算解密矩阵。

解 计算方法为 $[\boldsymbol{A} | \boldsymbol{I}] \rightarrow [\boldsymbol{I} | \boldsymbol{A}^{-1}]$。

$$\left[\begin{array}{ccc|ccc}1&2&3&1&0&0\\0&1&4&0&1&0\\5&6&0&0&0&1\end{array}\right]\xrightarrow{-5r_1+r_3}\left[\begin{array}{ccc|ccc}1&2&3&1&0&0\\0&1&4&0&1&0\\0&-4&-15&-5&0&1\end{array}\right]$$

$$\xrightarrow{4r_2+r_3}\left[\begin{array}{ccc|ccc}1&2&3&1&0&0\\0&1&4&0&1&0\\0&0&1&-5&4&1\end{array}\right]$$

$$\xrightarrow{-2r_2+r_1}\left[\begin{array}{ccc|ccc}1&0&-5&1&-2&0\\0&1&4&0&1&0\\0&0&1&-5&4&1\end{array}\right]$$

$$\xrightarrow[-4r_3+r_2]{5r_3+r_1}\left[\begin{array}{ccc|ccc}1&0&0&-24&18&5\\0&1&0&20&-15&-4\\0&0&1&-5&4&1\end{array}\right],$$

于是得到解密矩阵

$$\boldsymbol{A}^{-1}=\begin{bmatrix}-24&18&5\\20&-15&-4\\-5&4&1\end{bmatrix}\equiv\begin{bmatrix}2&18&5\\20&11&22\\21&4&1\end{bmatrix}\bmod 26。$$

示例 2-4 中 a_{11} 正好为 1，若为非 1 整数，则需要计算 a_{11} 的乘法逆元，将 a_{11} 转换为 1，这样可以在后续计算中避免计算过多的乘法逆元而降低计算速度。

2.4.4 希尔密码的实现

希尔密码的实现要解决以下几个问题：

(1) 在得到加密密钥后需计算解密密钥。

(2) 在计算解密密钥时需要计算行列式的乘法逆元。

(3) 在加密过程中需要将明文分割成与加密矩阵的阶相同大小的小段。

(4) 在解密过程中同样需要将密文分割成与解密矩阵的阶同样的大小的小段。

在这一节的示例中假设 26 个小写字母与数字形成映射，在需加密的文件或数据中不包含其他字符。采用二阶矩阵作为加密矩阵。

程序主要由 Hill 类来完成数据定义以及对数据的操作，类及相关结构体的声明见程序清单 2-14。

程序清单 2-14

```
struct ExtEuc
{
    int d;
    int s;
    int t;
};
class Hill
{
    private:
        int key[2][2];                          //加密密钥
```

```
        int decKey[2][2];                   //解密密钥
        int det;                            //密钥的行列式
        string plainText;                   //加密前明文
        vector< char> cipherText;           //加密后密文
        vector< char> vinText;              //临时存储明文数据
        vector< char> decText;              //解密后的文件数据
    public:
        Hill();
        void init();                        //初始化密钥、明文
        void cutPlainText();                //分割明文存储到向量
        void getDecKey();                   //获取解密密钥
        void encryption();                  //加密明文
        void decryption();                  //解密密文
        void showResult();                  //显示计算结果
        int gcd(int n,int k);               //计算最大公因子
        ExtEuc ExtEuclid(int,int);          //用于计算乘法逆元的函数
};
```

结构体 ExtEuc 主要用于计算行列式的乘法逆元,其他各成员变量和成员函数的功能参照注释中的说明。

初始化加密密钥、明文,计算解密密钥等在初始化函数 init()中完成,其代码见程序清单 2-15。

程序清单 2-15

```
void Hill::init()
{
    int i,j;
    while(1)
    {
        cout<< "Input the key:"<< endl;
        for(i= 0;i< 2;i+ + )
        {
            for(j= 0;j< 2;j+ + )
            {
                cin> > key[i][j];
            }
        }
        det= key[0][0] * key[1][1]- key[0][1] * key[1][0];
        if((det= = 0) | (gcd(det,26) != 1))
        {
            cout<< "The key can't be inversed, reinput the key!"<< endl;
        }
        else
        {
            break;
```

```
22            }
23        }
24        getDecKey();
25        cout<<"Input the plaintext:"<<endl;
26        cin>>plainText;
27    }
```

程序清单 2-15 中，4～23 行处理加密密钥，其中的 14～18 行判断加密矩阵是否为可逆矩阵，若不可逆则重新输入加密矩阵，同时 det 的值是否符合解密的判断要求（与字符集的元素个数互素），若不符合要求同样需重新输入加密矩阵。在判断过程中使用 gcd()函数，用于计算最大公因子，具体实现方法见程序清单 2-6。第 24 行调用 getDecKey()函数，计算解密密钥，第 26 行输入需要加密的明文。

init()函数直接由构造函数调用，完成初始化数据的任务。

计算解密密钥的函数 getDecKey()见程序清单 2-16。

程序清单 2-16

```
01    void Hill::getDecKey()
02    {
03        ExtEuc eu;
04        int inverseDet;                    //det 的乘法逆元
05        int i,j;
06        eu=ExtEuclid(det,26);
07        inverseDet=eu.s;                   //获得 det 的乘法逆元
08        decKey[0][0]=key[1][1];
09        decKey[0][1]=-key[0][1];
10        decKey[1][0]=-key[1][0];
11        decKey[1][1]=key[0][0];
12        for(i=0;i<2;i++)
13        {
14            for(j=0;j<2;j++)
15            {
16                decKey[i][j]=(decKey[i][j]*inverseDet)%26;
17                if(decKey[i][j]<0)
18                {
19                    decKey[i][j]+=26;
20                }
21            }
22        }
23    }
```

程序清单 2-16 中的第 6 行和第 7 行计算行列式的乘法逆元，ExtEuclid()函数的内容与程序清单 2-8 相同，第 12～22 行计算解密密钥，第 19 行消除解密密钥中存在的负值。

函数 cutPlainText()的功能为将输入的明文分解成字符，并存储到向量中，目的是方便后续加密和解密的处理。代码见程序清单 2-17。

程序清单 2-17

```
01 void Hill::cutPlainText()
02 {
03     int i;
04     if(plainText.length()%2==1)
05     {
06         plainText+="z";
07     }
08     for(i=0;i<plainText.length();i++)
09     {
10         vinText.push_back(plainText[i]);
11     }
12 }
```

函数 cutPlainText()包括两部分的内容,第 4～7 行代码用于判断明文长度的奇偶性。若明文长度为奇数,则在明文字符的最后加一个字符“z”,以便与二阶加密矩阵的阶数相匹配。第 8～11 行将明文分解为字符串,并添加到向量。

函数 cutPlainText()的功能也可以直接在加密函数中实现,但会使程序的可读性变差。

函数 encryption()的功能为完成对明文的加密,代码见程序清单 2-18。

程序清单 2-18

```
void Hill::encryption()
{
    cutPlainText();
    int ciphText[2]={0,0};          //处理临时密文数据
    int plaiText[2]={0,0};          //处理临时明文数据
    int i,j,k;
    for(i=0;i<vinText.size();i+=2)
    {
        plaiText[i%2]=int(vinText[i])-int('a');
        plaiText[(i+1)%2]=int(vinText[i+1])-int('a');
        for(j=0;j<2;j++)
        {
            for(k=0;k<2;k++)
            {
                ciphText[j]+=key[j][k]*plaiText[k];
            }
        }
        for(j=0;j<2;j++)
        {
            ciphText[j]=ciphText[j]%26;
            cipherText.push_back(char(ciphText[j]+int('a')));
            ciphText[j]=0;          //还原临时数据
```

```
23            }
24        }
25    }
```

程序清单 2-18 中的第 3 行分割明文。第 7 行到第 24 行加密明文，其中在第 7 行的循环处理中，每次处理两个字符，第 9 行和第 10 行将明文字符转换为整形数字，第 11 行到第 17 行加密明文，第 18 行到第 23 行将整形数据的密文转换为字符，并加入到向量，第 22 行还原密文临时数据。

函数 void decryption()的功能为解密密文数据，代码见程序清单 2-19。

程序清单 2-19

```
01    void Hill::decryption()
02    {
03        int ciphText[2]={0,0};          //处理临时密文数据
04        int plaiText[2]={0,0};          //处理临时明文数据
05        int i,j,k;
06        for(i=0;i<cipherText.size();i+=2)
07        {
08            ciphText[i%2]=int(cipherText[i])-int('a');
09            ciphText[(i+1)%2]=int(cipherText[i+1])-int('a');
10            for(j=0;j<2;j++)
11            {
12                for(k=0;k<2;k++)
13                {
14                    plaiText[j]+=decKey[j][k]*ciphText[k];
15                }
16            }
17            for(j=0;j<2;j++)
18            {
19                plaiText[j]=plaiText[j]%26;
20                decText.push_back(char(plaiText[j]+int('a')));
21                plaiText[j]=0;
22            }
23        }
24    }
```

程序清单 2-19 中的第 6 行到第 23 行实现解密过程，其中，第 8 行和第 9 行将密文字符转换为整形数字，第 10 行到第 16 行为解密计算过程，第 17 行到第 22 行将解密得到的明文数字转换为字符，其中第 21 行还原临时明文数据。

2.5 习题与实践题

2.5.1 习题

1. 简要说明单表代替密码算法的基本原理。

2. 简要说明移位密码算法的基本原理。

3. 已知 $xy=1 \bmod z$,并且有 $x=7,z=23$,试计算乘法逆元 y。

4. 简要说明乘数密码算法的加密和解密基本原理,并举例说明。

5. 简要说明多表代替密码算法的基本原理,并简要说明单表代替密码算法与多表代替密码算法之间的异同。

6. 已知希尔密码的加密矩阵为 $\begin{bmatrix} a & b \\ c & d \end{bmatrix}=\begin{bmatrix} 5 & 4 \\ 6 & 7 \end{bmatrix}$,并且字母映射表为 26 个字母的映射表,问:该加密矩阵是否适合?若适合则计算解密矩阵。

2.5.2 实践题

1. 单表代替密码算法的明文既可以来源于文件,也可以来源于控制台,假设由 26 个字母与空格组成单表代替密码,试修改 2.1 节中单表代替密码算法的实现过程,将明文数据通过控制台输入,并对输入的明文进行加密,将加密的结果在控制台输出,然后再对加密后的密文进行解密,将解密后的密文在控制台进行输出。(注意事项:空格也需要读入,同时也需要对空格进行加密和解密。)

2. 凯撒密码算法是移位算法的一种特殊情况,明文由 26 个字母组成,密文由与明文相同的 26 个字母组成,试编写一程序完成凯撒密码算法的加密与解密,明文数据通过文件读取,加密后的密文保存到文件,解密是从文件读取密文,并将解密后的数据保存到相应的文件,同时根据实际加密和解密过程分析使用 26 个字母进行加密和解密所存在的问题。

3. 在 2.3.4 节中,扩展的欧几里得算法通过一个结构体来处理计算过程中的中间变量,试改造该节中的实现方法,将函数的返回值直接改为返回乘法逆元。

4. 在 2.4.4 节中的希尔密码算法的实现过程中使用了向量来处理加密和解密的数据,试改造该程序,使用数组的方法来处理明文和密文。

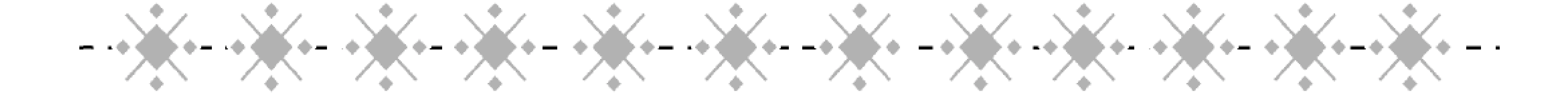
第 2 部分

现代对称密码

对称密码是指加密密钥与解密密钥相同或能推算的加密算法，其特点是解密密钥能够从加密密钥中推算出来，加密密钥也能够从解密密钥中推算出来。在大多数对称算法中，加密密钥和解密密钥是相同的，此时，这种算法也被称为单密钥算法。对称算法的安全性依赖于密钥，密钥泄露意味着任何人都可以对它们发送或接收的消息进行解密。

对称加密算法的优点在于加解密的高速度和在使用长密钥时的难破解性，但密钥的生成和分发是使用对称密码算法的最大问题。

目前在国际上使用的对称密码算法主要有：DES 算法、三重 DES 算法、CAST-128 算法、Blowfish 算法、RC5 算法、IDEA 算法等。



第 3 章

S-DES 算法

S-DES 算法又称为简化的 DES 算法，是美国圣达拉卡大学的爱德华·施菲尔教授提出的用于教学的一个算法，其结构和性质与 DES 算法相似。学习 S-DES 算法有助于加深理解 DES 算法。

3.1 S-DES 算法原理

S-DES 整体结构见图 3-1。

S-DES 加密算法的输入是一个 8 位的明文组（例如 10111101）和一个 10 位的密钥，输出为 8 位的密文组。S-DES 解密算法的输入是一个 8 位的密文组和一个 10 位的密钥，输出为 8 位的明文组。

S-DES 的加密算法包括 5 步：通过 IP 函数进行初始置换、通过复杂函数 f_K 进行置换和代换运算（这部分运算依赖于输入的密钥）、通过 SW 函数进行两部分的简单置换、再次通过 f_K 函数进行置换和代换、通过置换函数 IP 的逆函数 IP^{-1} 进行置换。S-DES 算法通过多层置换和代换操作来增加密码分析的难度。

S-DES 的加密过程可以简单地表示为

$$IP^{-1} \cdot f_{K_2} \cdot SW \cdot f_{K_1} \cdot IP。\quad (3\text{-}1)$$

也可以写为

$$密文 = IP^{-1}(f_{K_2}(SW(f_{K_1}(IP(明文)))))。\quad (3\text{-}2)$$

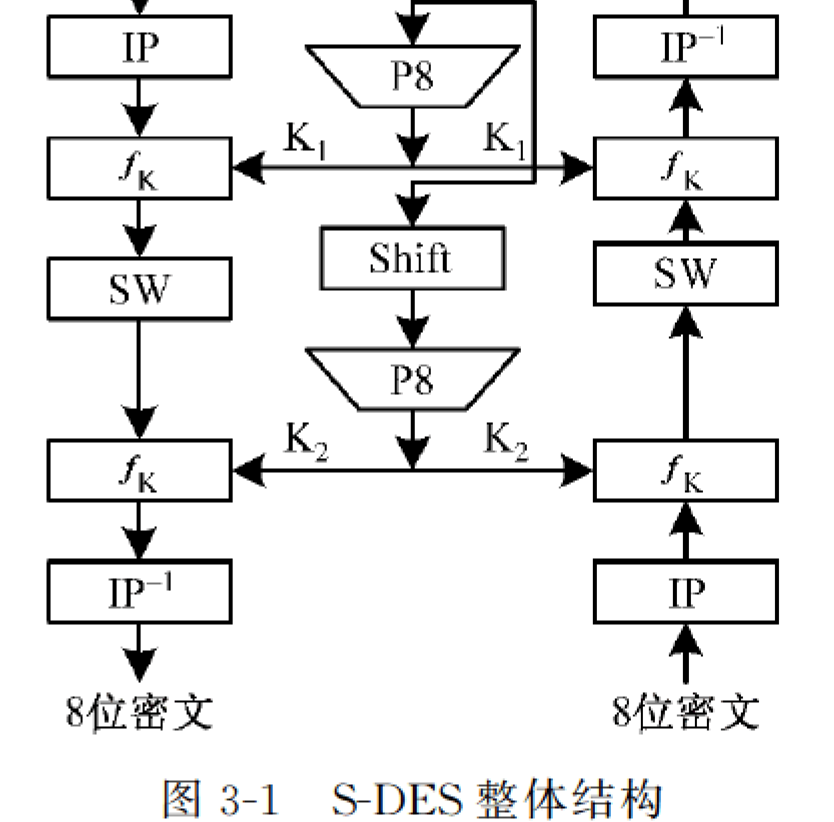


图 3-1　S-DES 整体结构

解密过程是加密过程的逆过程，可以表示为

$$明文 = IP^{-1}(f_{K_1}(SW(f_{K_2}(IP(密文)))))。\quad (3\text{-}3)$$

3.2 S-DES 密钥的生成

在 S-DES 的加密和解密过程中分别使用了两次 f_K 运算，f_K 运算过程中分别使用了不同的两个密钥，K_1 和 K_2。K_1 和 K_2 的生成的基本过程可以表示为

$$K_1 = P8(Shift(P10(key))),$$
$$K_2 = P8(Shift(Shift(P10(key))))。 \quad (3-4)$$

密钥生成的基本过程如图 3-2 所示。

将 10 位的密钥表示成$(k_1,k_2,k_3,k_4,k_5,k_6,k_7,k_8,k_9,k_{10})$,那么 P10 置换定义为

$$P10(k_1,k_2,k_3,k_4,k_5,k_6,k_7,k_8,k_9,k_{10}) = (k_3,k_5,k_2,k_7,k_4,k_{10},k_1,k_9,k_8,k_6)$$

或可以按下面形式定义为

P10									
3	5	2	7	4	10	1	9	8	6

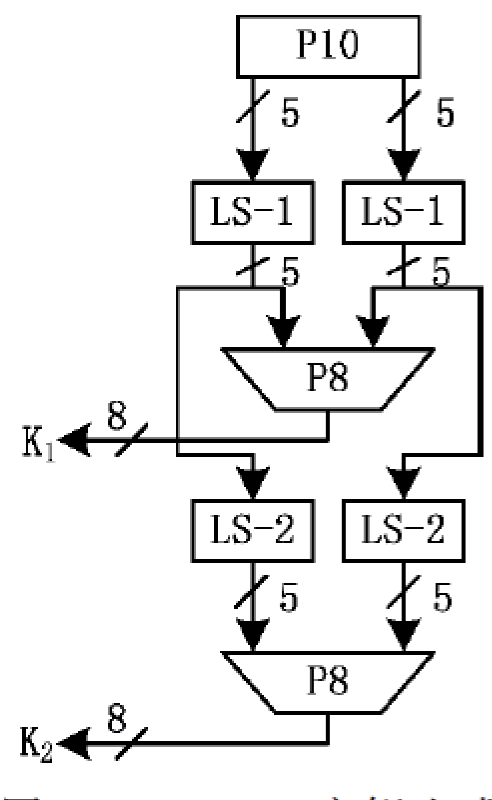


图 3-2 S-DES 密钥生成

即第 1 个位置输出的是第 3 位,第 2 个位置输出的是第 5 位,以此类推。例如,初始的密钥为(10100 00010),经过置换后变成了(10000 01100),接下来分别对前面 5 位和后面 5 位循环左移(LS-1)1 位,密钥变为(00001 11000)。然后按下面的 P8 置换规则计算子密钥 K_1,得到 K_1=(1010 0100)。

P8							
6	3	7	4	8	5	10	9

在对密钥进行循环左移 1 位后,得(00001 11000),再循环左移 2 位,那么,密钥(00001 11000)就转换为(00100 00011),再进行 P8 置换,得到另一个子密钥 K_2=(0100 0011)。

示例 3-1 假设 10 位 S-DES 密钥为 11000 11110,试计算子密钥 K_1 和 K_2。

解 K_1 生成过程为

Bit#	1	2	3	4	5	6	7	8	9	10
K	1	1	0	0	0	1	1	1	1	0
P10(K)	0	0	1	1	0	0	1	1	1	1
Shift(P10(K))	0	1	1	0	0	1	1	1	1	0
P8(Shift(P10(K)))	1	1	1	0	1	0	0	1		

得到 K_1=(1110 1001)。

K_2 的生成过程为

Bit#	1	2	3	4	5	6	7	8	9	10
K	1	1	0	0	0	1	1	1	1	0
P10(K)	0	0	1	1	0	0	1	1	1	1
Shift(P10(K))	1	0	0	0	1	1	1	0	1	1
P8(Shift(P10(K)))	1	0	1	0	0	1	1	1		

得到 K_2=(1010 0111)。

3.3 S-DES 加密与解密过程

S-DES 的加密过程可以简单表示为

$$密文 = IP^{-1}(f_{K_2}(SW(f_{K_1}(IP(明文)))))。$$

在加密过程中，两次进行置换操作，两次进行 f_K 运算，一次进行 SW 交换，其中核心部分是两次 f_K 运算。

S-DES 的加密过程如图 3-3 所示。

S-DES 算法的第 1 步是对输入的 8 位明文使用 IP 函数进行置换，其置换方法为

IP							
2	6	3	1	4	8	5	7

这个置换过程保留了所有 8 位明文的特性，但进行了混淆。

最后的置换使用了 IP^{-1} 函数进行置换，其置换方法为

IP^{-1}							
4	1	3	5	7	2	8	6

第 2 个置换是第 1 个置换的逆，即两个置换的关系为 $IP^{-1}(IP(X))=X$。

f_K 函数是 S-DES 算法中最为复杂的部分，它是由置换函数和代换函数组合而成。函数可以表达为：设 L 和 R 分别是 f_K 的 8 位输入的左边 4 位和右边 4 位，F 是一个 4 位串到 4 位串的映射（不一定是一对一映射）。那么设

$$f_K(L,R) = (L \oplus F(R,S_k),R) \qquad (3\text{-}5)$$

其中 S_k 是子密钥，$\oplus$是按位异或。

函数 $f_K(L,R)$的计算过程中，F 函数最为复杂，F 函数的执行过程如下：

(1) F 函数中的 R 是明文经过 IP 置换后得到的输出的右半部分，F 函数首先执行对 R 的 E/P 扩展置换，将 4 位 R 扩展置换成 8 位，其扩展置换的方法为

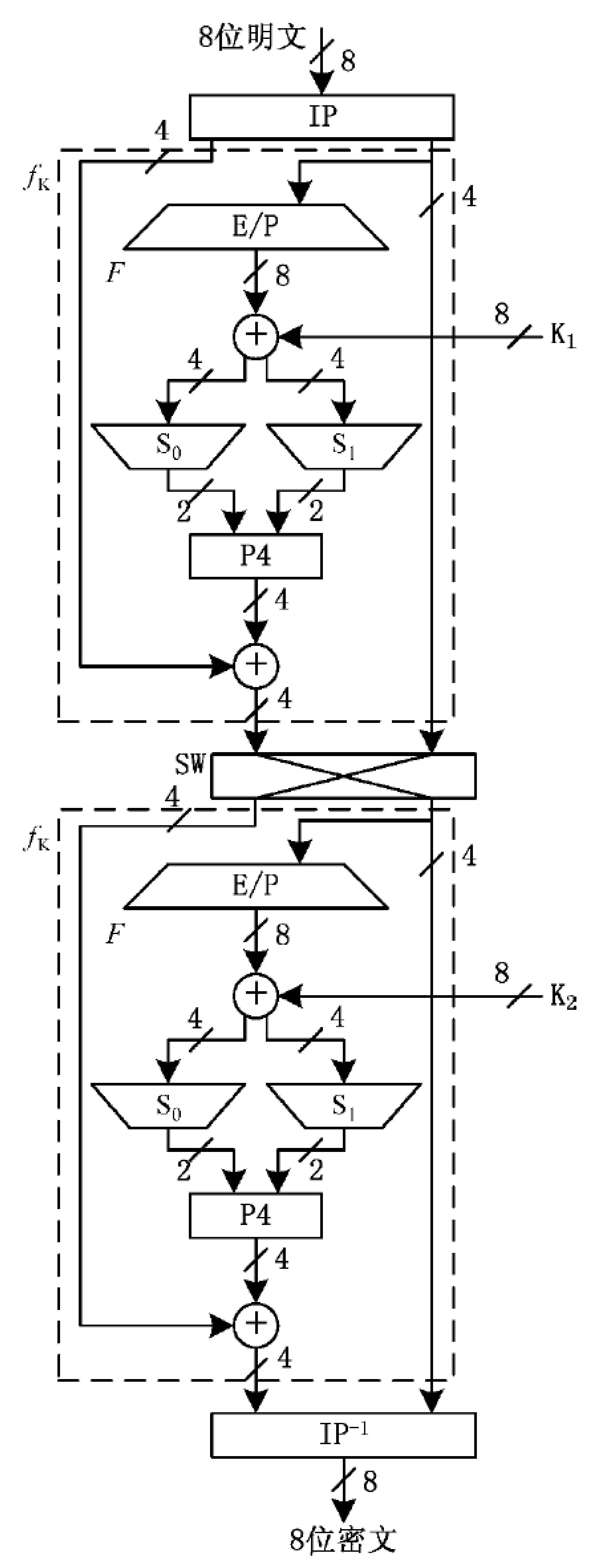


图 3-3 S-DES 算法的加密过程

E/P							
4	1	2	3	2	3	4	1

(2) 将 E/P 扩展置换得到的结果与密钥进行 XOR 运算。

(3) 根据 S 盒对第 2 步获得的数据分为左、右两部分进行替换。S 盒的定义为

$$S_0=\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 0 & 3 & 2 \\ 1 & 3 & 2 & 1 & 0 \\ 2 & 0 & 2 & 1 & 3 \\ 3 & 3 & 1 & 3 & 2 \end{array},\quad S_1=\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 2 & 3 \\ 1 & 2 & 0 & 1 & 3 \\ 2 & 3 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 & 3 \end{array}。\qquad (3\text{-}6)$$

假设从第 2 步中获得的结果为：$P_{0.0}\ P_{0.1}\ P_{0.2}\ P_{0.3}\ P_{1.0}\ P_{1.1}\ P_{1.2}\ P_{1.3}$，$S_0$ 盒与左边 4 位进行运算，S_1 盒与右边 4 位进行运算。运算方法为：第 1 位和第 4 位决定 $S_{盒}$ 的行，第 2 位和第 3 位决定 S 盒的列。例如：有$(P_{0.0}\ P_{0.3})=(00)$，$(P_{0.1}\ P_{0.2})=(10)$，那么，输出的是 S_0 盒的第 0 行第 2 列，结果为 3，写成二进制则为 11。同理，可以通过$(P_{1.0}\ P_{1.3})$、$(P_{1.1}\ P_{1.2})$与 S_1 盒产生另一个 2 位的二进制输出，将两者结合得到 4 位的二进制输出。

（4）将 S 盒的输出进行 P4 置换，P4 置换的方法为

P4			
2	4	3	1

P4 函数的输出就是 F 函数的输出。

示例 3-2 假设明文为 0010 1000，密钥 $K_1=(1110\ 1001)$，试计算 f_{K_1}。

解 （1）首先进行 IP 置换，置换过程为

Bit#	1	2	3	4	5	6	7	8
IP	2	6	3	1	4	8	5	7
P	0	0	1	0	1	0	0	0
IP(P)	0	0	1	0	0	0	1	0

得到：IP(P)=(00100010)

（2）$f_{K_1}(L,R)=f\{11101001\}(0010,0010)=(0010\oplus F(0010,\{1110\ 1001\}),0010)$

（3）$F(0010,\{1110\ 1001\})=P4\cdot \text{SBoxes}\cdot\{1110\ 1001\}\oplus(E/P(0010))$

（4）完整计算过程如下：

Bit#	1	2	3	4	5	6	7	8
R	0	0	1	0				
E/P(R)	0	0	0	1	0	1	0	0
K_1	1	1	1	0	1	0	0	1
$E/P(R)\oplus K_1$	1	1	1	1	1	1	0	1
$\text{SBoxes}(E/P(R)\oplus K_1)$	1	0	0	0				
$P4(\text{SBoxes}(E/P(R)\oplus K_1))$	0	0	0	1				

（5）运行函数 F 后得到的结果为 0001。

（6）计算 $f_{K_1}(L,R)=(0010\oplus 0001,0010)=(0011,0010)$。

在执行完 $f_{K_1}(L,R)$函数之后执行 SW 交换函数，SW 交换函数的功能是交换输入的左半部分与右半部分，这样 $f_{K_2}(L,R)$函数就作用在不同的部分了。在 f_{K_2} 的执行过程中，E/P、S_0、S_1 和 P4 都是相同的，但输入的密钥为 K_2。

示例 3-3 根据示例 3-1 和示例 3-2 得到的计算结果，计算加密后的明文。

解 (1) 在示例 3-1 中,计算得到 K_2=(1010 0111)。在示例 3-2 中计算得到的 f_{K_1}(L,R)=(0011,0010),因此有:L=0011,R=0010,进行 SW 交换后有:L=0010,R=0011。

(2) 计算 $f_{K_2}(L,R)=f\{10100111\}(0010\ 0011)=(0010\oplus F(0011,\{1010\ 0111\}),0011)$。

(3) 计算 F 函数,F 函数计算过程如下:

Bit#	1	2	3	4	5	6	7	8
R	0	0	1	1				
E/P(R)	1	0	0	1	0	1	1	0
K_2	1	0	1	0	0	1	1	1
E/P(R)$\oplus K_2$	0	0	1	1	0	0	0	1
SBoxes(E/P(R)$\oplus K_2$)	1	0	1	0				
P4(SBoxes(E/P(R)$\oplus K_2$))	0	0	1	1				

(4) 此时我们得到 F 为(0011)。

(5) 计算 $f_{K_2}(L,R)=f\{10100111\}(0010\oplus 0011,0011)=(0001,0011)$。

(6) 最后计算 IP^{-1} 置换,计算过程为

Bit#	1	2	3	4	5	6	7	8
R,L	0	0	0	1	0	0	1	1
IP^{-1}(R,L)	1	0	0	0	1	0	1	0

(7) 最终加密结果为:1000 1010。

S-DES 算法的解密过程如下:

$$明文 = IP^{-1}(f_{K_1}(SW(f_{K_2}(IP(密文)))))$$

具体实现方法的各个步骤与加密方法的具体实现完全一样。

3.4 S-DES 算法实现

S-DES 算法的实现过程中主要需解决以下几个问题:

(1) 置换运算;

(2) 字符左右两半的交换运算;

(3) E/P 扩展;

(4) S 盒运算;

(5) 移位运算。

以上各部分的运算方法将在程序实现的过程中给出具体的解释。

S-DES 算法实现的主要结构如图 3-4 所示。

变量与函数封装在 SDES 类中,类的代码见程序清单 3-1。

程序清单 3-1

```
class SDES
{
    public:
```

SDES	
- key	: int
- key1	: int
- key2	: int
- plainText	: int
- cipherText	: int
- decipherText	: int
- P10[10]	: static const int
- P8[8]	: static const int
- IP[8]	: static const int
- IPI[8]	: static const int
- EP1[4]	: static const int
- EP2[4]	: static const int
- S0[4][4]	: static const int
- S1[4][4]	: static const int
+ SDES ()	
+ setKey (string s)	: void
+ setPlainText (string s)	: void
+ fk (int key, int& left, int right)	: void
+ getKey ()	: void
+ encryption ()	: void
+ decryption ()	: void
+ permutations (int num, const int p[], int pmax, int n)	: int
+ shiftLS (int num)	: int
+ showResult ()	: void
- stringToInt (string s)	: int

图 3-4　S-DES 实现程序主要结构

```
        SDES();
        void setKey(string s);                          //设置密钥
        void setPlainText(string s);                    //设置明文
        void fk(int key,int &left,int right);           //fK 运算
        void getKey();                                  //获得密钥 key1 和 key2
        void encryption();                              //加密过程
        void decryption();                              //解密过程
        int permutations(int num,const int p[ ],int pmax,int n);    //进行置换操作
        int shiftLS(int num);                           //循环左移 (左右各半分别左移 1 位)
        void showResult();                              //显示结果
    private:
        int key;                                        //初始密钥
        int key1;                                       //密钥 K1
        int key2;                                       //密钥 K2
        int stringToInt(string s);                      //将输入的字符串 (0、1)转换为整型数据
        int plainText;                                  //明文
        int cipherText;                                 //密文
        int decipherText;                               //解密后的明文
        //以下为常量
        static const int P10[10];                       //P10 置换常量
        static const int P8[8];                         //P8 置换常量
        static const int IP[8];                         //IP 置换常量
        static const int IPI[8];                        //IP-1置换常量
        static const int EP1[4];                        //E/P1 置换常量
```

```
        static const int EP2[4];                    //E/P2 置换常量
        static const int P4[4];                     //P4 置换常量
        static const int S0[4][4];                  //S0 盒
        static const int S1[4][4];                  //S1 盒
};
```

程序实现过程为：输入密钥和明文、计算密钥 K_1 和 K_2、加密明文、解密密文。实现过程中明文和密文都是用整型数据类型来处理，并且根据需要仅处理相应的位数，例如：明文和密文在处理时仅处理 8 位数据，其他位数都置为 0。输入的密钥为字符型数据，在转换成整型数据之后再用于密钥 K_1 和 K_2 的计算。各个常量的值单独初始化，其初始化过程见程序清单 3-2。

程序清单 3-2

```
const int SDES::P10[10]={3,5,2,7,4,10,1,9,8,6};
const int SDES::P8[8]={6,3,7,4,8,5,10,9};
const int SDES::IP[8]={2,6,3,1,4,8,5,7};
const int SDES::IPI[8]={4,1,3,5,7,2,8,6};
const int SDES::EP1[4]={4,1,2,3};
const int SDES::EP2[4]={2,3,4,1};
const int SDES::P4[4]={2,4,3,1};
const int SDES::S0[4][4]={{1,0,3,2},{3,2,1,0},{0,2,1,3},{3,1,3,2}};
const int SDES::S1[4][4]={{0,1,2,3},{2,0,1,3},{3,0,1,0},{2,1,0,3}};
```

密钥生成过程是先将输入的字符串密钥转换为整型数据，通过函数 setKey(string s)来获取密钥。在函数实现过程中通过另一个函数 stringToInt(string s)将字符串转换为整型数据，具体实现过程见程序清单 3-3。

程序清单 3-3

```
void SDES::setKey(string s)
{
    key=stringToInt(s);
}
int SDES::stringToInt(string s)
{
    int key=0;              //整型表示的密钥
    int i;
    int n=s.length();       //n 为字符串的长度
    for(i=0;i<n;i++)
    {
        key=key*2+s[i]-'0';
    }
    return key;
}
```

在获取密钥之后，通过函数 getKey()来计算子密钥 key1 和 key2，getKey()函数的代码见程序清单 3-4。

程序清单 3-4

```
01 void SDES::getKey()
02 {
03     int tempKey;
04     tempKey=permutations(key,P10,10,10);
05     tempKey=shiftLS(tempKey);
06     key1=permutations(tempKey,P8,10,8);
07     tempKey=shiftLS(tempKey);
08     tempKey=shiftLS(tempKey);
09     key2=permutations(tempKey,P8,10,8);
10 }
```

子密钥的计算过程与 3.1.2 节所述的计算过程一致。在 getKey()函数中通过调用函数 permutations(int num, const int p[], int pmax,int n)来实现置换操作,通过调用函数 shiftLS(int num)来实现循环移位操作。函数 permutations(int num, const int p[], int pmax,int n)完整代码见程序清单 3-5。

程序清单 3-5

```
int SDES::permutations(int num,const int p[ ],int pmax,int n)
{
    int temp=0;
    int i;
    for(i=0;i<n;i++)
    {
        temp<<=1;          //temp 左移一位
        temp|=(num>>(pmax-p[i]))&1;
    }
    return temp;
}
```

函数参数 num 为输入,p[]为置换矩阵,pmax 为 num 要处理的位数,n 为输出的大小(位数)。函数的实现利用了位运算的特征。处理过程的基本思想为:将要处理的第 n 位数据移到最右侧,与 1 进行“&”运算,若最后一位是 0,那么整个结果为 0,否则为 1。然后与 temp 进行“或”运算,由于 temp 先进行了左移,最后一位为 0,因此运算结果取决于“&1”运算的结果。使用位运算方法,有效简化了置换操作。假设输入的密钥为:11000 11110,程序清单中第 4 行调用 permutations(key,P10,10,10),其中 P10[10]={3,5,2,7,4,10,1,9,8,6},在第 1 轮运算中有

num	1	1	0	0	0	1	1	1	1	0
temp	0	0	0	0	0	0	0	0	0	0
temp<<1	0	0	0	0	0	0	0	0	0	0
num>>(pmax−3)	0	0	0	0	0	0	0	1	1	0
(num>>(pmax−3))&1	0	0	0	0	0	0	0	0	0	0
temp|((num>>(pmax−3))&1)	0	0	0	0	0	0	0	0	0	0

在第 2 轮运算中有

num	1	1	0	0	0	1	1	1	1	0
temp	0	0	0	0	0	0	0	0	0	0
temp<<1	0	0	0	0	0	0	0	0	0	0
num>>(pmax-5)	0	0	0	0	0	1	1	0	0	0
(num>>(pmax-5))&1	0	0	0	0	0	0	0	0	0	0
temp|((num>>(pmax-5))&1)	0	0	0	0	0	0	0	0	0	0

在第 3 轮运算中有

num	1	1	0	0	0	1	1	1	1	0
temp	0	0	0	0	0	0	0	0	0	0
temp<<1	0	0	0	0	0	0	0	0	0	0
num>>(pmax−2)	0	0	0	0	0	0	0	0	1	1
(num>>(pmax−2))&1	0	0	0	0	0	0	0	0	0	1
temp|((num>>(pmax−2))&1)	0	0	0	0	0	0	0	0	0	1

在第 4 轮运算中有

num	1	1	0	0	0	1	1	1	1	0
temp	0	0	0	0	0	0	0	0	0	1
temp<<1	0	0	0	0	0	0	0	0	1	0
num>>(pmax−7)	0	0	0	1	1	0	0	0	1	1
(num>>(pmax−7))&1	0	0	0	0	0	0	0	0	0	1
temp|((num>>(pmax−7))&1)	0	0	0	0	0	0	0	0	1	1

以此类推可获得最终结果：00110 01111。

函数 permutations(int num，const int p[]，int pmax,int n)实现了 S-DES 中所需的各种置换操作。

程序清单 3-4 中用到 shiftLS(int num)来实现循环移位操作，函数参数为输入，shiftLS(int num)函数实现的是左右两部分各循环左移一位的功能，完整代码见程序清单 3-6。

程序清单 3-6

```
int SDES::shiftLS(int num)
{
      int temp;
      int L,R;                        //左右各半数据
      L=(num>>5)&0x1F;
      R=num&0x1F;
      L=((L&0xF)<<1)|((L&0x10)>>4);
      R=((R&0xF)<<1)|((R&0x10)>>4);
      temp=(L<<5)|R;
      return temp;
}
```

对输入数据的左右两部分进行循环左移同样利用了位运算的特征。例如，在第 5 行中计算左边数据时是将输入右移 5 位，假如输入为 00110 01111，那么计算 L 的右移过程为

num	0	0	1	1	0	0	1	1	1	1
(num>>5)	0	0	0	0	0	0	0	1	1	0
(num>>5)&0x1F	0	0	0	0	0	0	0	1	1	0

1F 的二进制为 11111，那么 &0x1F 的作用是将除右侧 5 位数据外，其他位置的数据均置为 0。例如，若输入的数据为 11111111，则与 0x1F 进行“&”运算的结果为 0001111。而计算 R 时则直接进行 &0x1F 运算，假如输入为 00110 01111，那么计算过程为

num	0	0	1	1	0	0	1	1	1	1
0x1F	0	0	0	0	0	1	1	1	1	1
num&0x1F	0	0	0	0	0	0	1	1	1	1

在获得左、右各部分的数据之后，要进行循环左移 1 位，其实现过程为输入的数据与 0xF 进行“&”运算后左移 1 位。将输入数据与 0x10 进行“&”运算，然后右移 4 位获得另一个数据，将两个数据进行“或”运算最终获得循环左移 1 位的结果。

例如，输入的数据为 10010，进行循环左移的过程如下：

num1	0	0	1	1	0
num1&(0xF)	0	0	1	1	0
(num1&(0xF))<<1	0	1	1	0	0

在第 1 个数据处理完之后，处理第 2 个数据：

num2	0	0	1	1	0
num2&(0x10)	0	0	0	0	0
(num2&(0x10))>>4	0	0	0	0	0

然后将两个数进行“或”运算，得 num1|num2＝01100。

在获得两个子密钥之后，就可以进行加密和解密。加密过程包括 IP 置换、f_{K_1} 运算、SW 左右交换运算、f_{K_2} 运算和 IP^{-1} 置换，在程序中通过函数 encryption()来实现。encryption()函数具体实现的代码见程序清单 3-7。

程序清单 3-7

```
void SDES::encryption()
{
    cipherText=permutations(plainText,IP,8,8);
    int R,L;                        //置换后密文分成左、右两半
    R=cipherText&0xF;
    L=((cipherText&0xF0)>>4);
    fk(key1,L,R);
    int temp=L;
    L=R;
    R=temp;
    fk(key2,L,R);
```

```
12        temp=(L<<4)|R;
13        cipherText=permutations(temp,IPI,8,8);
14    }
```

在程序清单 3-7 中，第 3 行是调用置换函数完成 IP 置换，第 4 行到第 6 行将 IP 置换得到的数据取右 8 位分割成左右各半，第 7 行为使用密钥 key1 进行的第一次 fk 运算，第 8 行到第 10 行为 SW 运算，第 11 行为使用密钥 key2 进行的第二次 fk 运算，第 13 行为 IP^{-1} 置换。

在整个 encryption()函数中，仅 fk 运算尚未解决，fk 运算的相关函数见程序清单 3-8。

程序清单 3-8

```
01  void SDES::fk(int key,int &left,int right)
02  {
03        int L=left;
04        int R=right;
05        int temp;
06        int tempL,tempR;
07        temp=((permutations(R,EP1,4,4))<<4)|(permutations(R,EP2,4,4));
08        temp=temp^key;
09        tempR=temp&0xF;
10        tempL=((temp&0xF0)>>4);
11        tempL=S0[((tempL&0x8)>>2)|(tempL&1)][(tempL>>1)&0x3];
12        tempR=S1[((tempR&0x8)>>2)|(tempR&1)][(tempR>>1)&0x3];
13        temp=(tempL<<2)|tempR;
14        temp=permutations(temp,P4,4,4);
15        left=L^temp;
16  }
```

在程序清单 3-8 中，第 7 行代码实现了 E/P 扩展，第 8 行代码实现 E/P 扩展后获得的输出与密钥进行异或运算，第 11 行和第 12 行是进行 S 盒运算，第 14 行是实现 P4 运算，第 15 行是“异或”运算。

在函数的参数中 int &left 使用了引用参数，这样，能直接获得 left 的值。

在 fk()函数中比较复杂的是 S 盒运算，S 盒由二维数组组成。行由第 1 位和第 4 位确定，列由第 2 位和第 3 位来确定，假设输入的数据是 1010，在执行“&0x8”运算后，除第 1 位数据保持不变外，其他数据置为 0，然后再右移 2 位计算如下：

tempL	1	0	1	0
0x8	1	0	0	0
tempL&0x8	1	0	0	0
(tempL&0x8)>>2	0	0	1	0

另一位数据只要和“1”进行“&”运算，这样仅保留最后一位数，其他均置为 0，将两者进行“|”运算后就获得“行”。

列运算是取第 2 位和第 3 位，先右移 1 位，再与“0x3”进行“&”运算，即获得“列”，若输入的数据为 1010，其运算过程如下：

tempL	1	0	1	0
0x3	0	0	1	1
tempL>>1	0	1	0	1
(tempL>>1)&0x3	0	0	0	1

S1 盒的运算过程与 S0 盒的运算过程相同，只是使用了输入的不同部分。

解密函数 decryption()与加密函数的实现过程类似，代码见程序清单 3-9。

程序清单 3-9

```
void SDES::decryption()
{
    decipherText=permutations(cipherText,IP,8,8);
    int R,L;                          //置换后密文分成左右两半
    R=decipherText&0xF;
    L=((decipherText&0xF0)>>4);
    fk(key2,L,R);
    int temp=L;
    L=R;
    R=temp;
    fk(key1,L,R);
    temp=(L<<4)|R;
    decipherText=permutations(temp,IPI,8,8);
}
```

decryption()函数中各部分的实现方法可参考加密函数 encryption()，具体实现的方法完全相同，只是在先后顺序上有所不同。

3.5 Feistel 密码结构

在对数据加密的方法中，比较典型的加密方法有两种，一种是流加密方法，该方法在加密过程中每次仅加密一位或一个字节，Vigenere 加密方法是典型的流加密方法。另一种加密方法是分组加密方法，这种方法在加密时将明文组作为整体进行加密，得到的密文通常是与之等长的密文组。

现在使用的几乎所有的对称分组加密算法都是基于 Feistel 分组密码结构的，因此，在进一步研究对称密码之前，了解 Feistel 密码结构是十分必要的。

Feistel 密码是以德国出生的、在 IBM 工作的物理学家及密码学家 Horst Feistel 命名的加密方法，属于分组对称密码结构。这种加密结构广泛应用于分组加密，包括数据加密标准(DES)算法。Feistel 结构的加密和解密过程非常相似，有时候除了密钥顺序有变化外，其他过程都一样。

Feistel 密码结构的加密和解密的基本结构如图 3-5 所示。

图 3-5 中，F 为轮函数(或圈函数)，K_0，K_1，…，K_n 为与轮数 0，1，…，n 相对应的子密钥。Feistel 密码结构的基本操作过程如下：

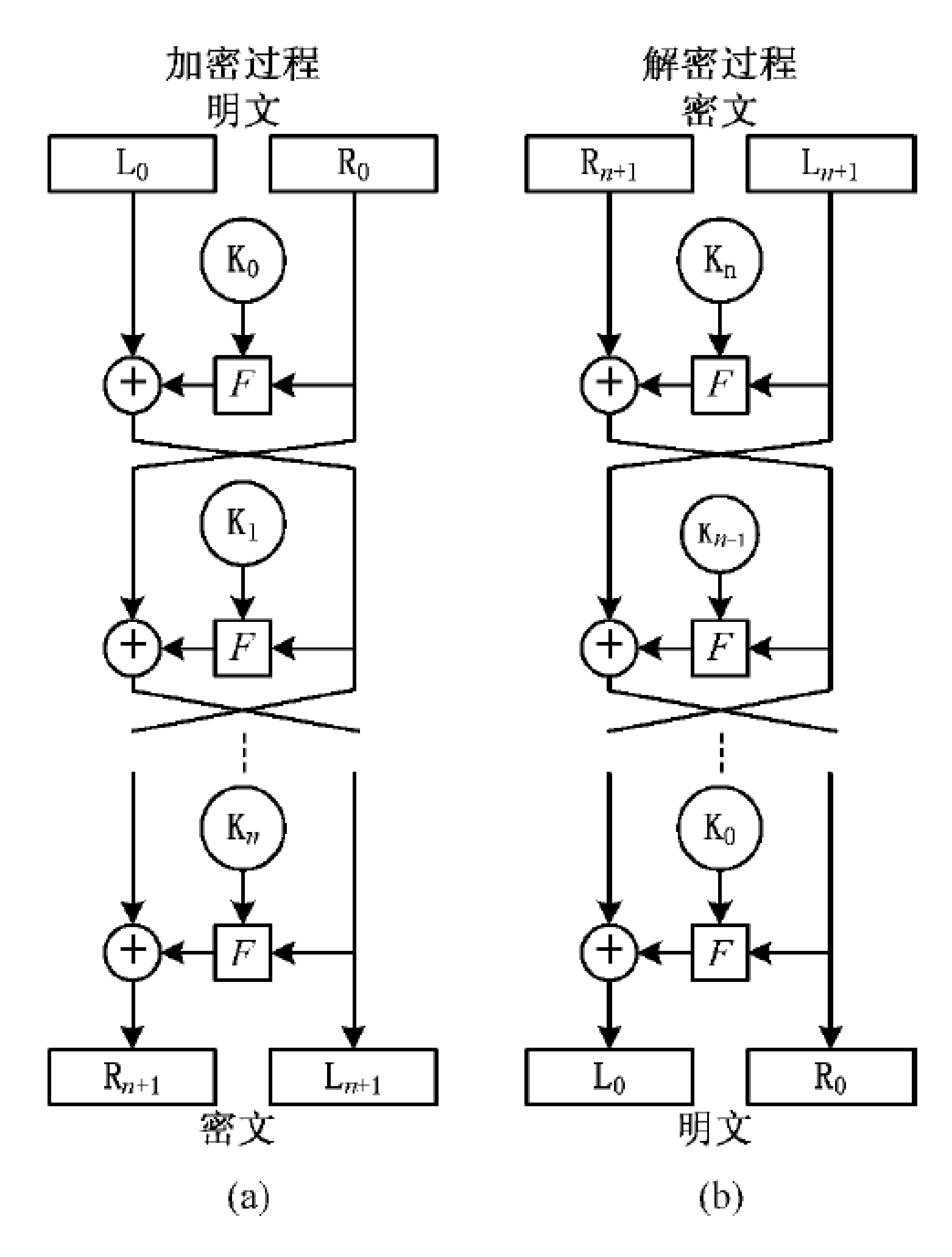


图 3-5 Feistel 密码结构

(1) 将输入的明文分割成相同大小的两部分(L_0,R_0)。

(2) 对 $i=0,1,\cdots,n$ 的每一轮计算:

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= L_i \oplus F(R_i, K_i) \end{aligned} \tag{3-7}$$

(3) 最后获得密文(R_{n+1},L_{n+1})。

解密密文(R_{n+1},L_{n+1})是计算 $i=n,n-1,\cdots,0$,每一轮的计算方法为

$$\begin{aligned} R_i &= L_{i+1} \\ L_i &= R_{i+1} \oplus F(L_{i+1}, K_i) \end{aligned} \tag{3-8}$$

最后获得的(L_0, R_0)就是解密得到的明文。

大多数采用 Feistel 结构的密码算法都是采用对称的方法,即左半部分和右半部分的大小相同,但也有少数加密算法采用了非对称的密码算法,Skipjack 加密算法就是一种非对称的加密算法。

3.6 习题与实践题

3.6.1 习题

1. 已知 S-DES 算法的输入密钥是 1011011011,试计算 S-DES 的密钥 K1 和 K2。

2. 在 S-DES 算法中使用了复杂函数 f_K 进行相应的运算,试简要说明 f_K 函数的基本原理,并给出相应的实例说明 f_K 函数的运算过程。

3. 已知输入的数据是 10111110,密钥 K1 为 10001011,试计算 f_{K_1}。

4. S-DES 算法中的一个重要核心技术是使用了 S 盒，试简要说明 S-DES 算法中 S 盒使用的基本原理，并通过相应的实例说明。

5. 简要说明 Feistel 密码结构的加密和解密的基本原理。

3.6.2 实践题

利用 3.4 节实现的 S-DES 算法，并对该实现方法进行适当改造，具体内容为：待加密的明文文件为“dataIn. txt”，密钥存储在“key. txt”文件中，加密后密文保存的文件为“cipher. txt”，经过解密后的文件存储在“decipher. txt”，同时程序具有比较明文和解密后明文相比较的功能，用于判断密文正常被解密，试完成上述程序。

提示：文件处理可以采用文件流，输入文件流和输出文件流既可以作为类的私有成员变量，也可以作为函数的相关参数来处理文件，在读取文件或输出文件时，可以使用输入流和输出流的重载来简化输入和输出。

第 4 章

DES 算 法

DES(Data Encryption Standard)曾是使用最广泛的数据加密标准,这个算法本身被称为数据加密算法(DEA)。DES 于 1977 年被美国国家标准局即现在的国家标准和技术研究所采纳为联邦信息处理标准 46(FIPS PUB 46)。

DES 目前被推荐在一般商业应用中使用,而不用 DES 来保护官方的数据,新版本的数据加密采用了三重 DES。因为 DES 与三重 DES 的加密、解密方法是相同的,因此理解 DES 算法仍有很重要的意义。

4.1 DES 算法原理

DES 算法的输入是 64 位明文,密钥长度为 64 位。在 64 位的密钥中,第 8、16、24、32、40、48、56 和 64 位是校验位,因此,DES 算法的实际密钥长度是 56 位。DES 算法的加密过程如图 4-1 所示。

DES 算法的基本过程可以描述如下:

(1) 在获得 64 位明文后,对明文进行一个初始置换,即 IP 置换。

(2) 对置换后的明文进行分组,分成左半部分和右半部分。

(3) 进行 16 轮完全相同的运算,这个运算的函数也被称为 F 函数。在每一轮的运算过程中使用各自不同的密钥。

(4) 在经过 16 轮运算后,将左右两部分合在一起,再经过一个末置换(初始置换的逆置换),这样就完成了算法的全部过程。

DES 的解密过程的算法与加密过程的算法完全一样,不同的地方有两点:第一个不同点是解密密钥的顺序与加密密钥正好相反,加密密钥分别是 $K_1, K_2, \cdots, K_{16}$,解密密钥分别是 $K_{16}, K_{15}, \cdots, K_1$。第二个不同点是子密钥产生时采用循环左移来产生子密钥,而解密密钥是通过循环

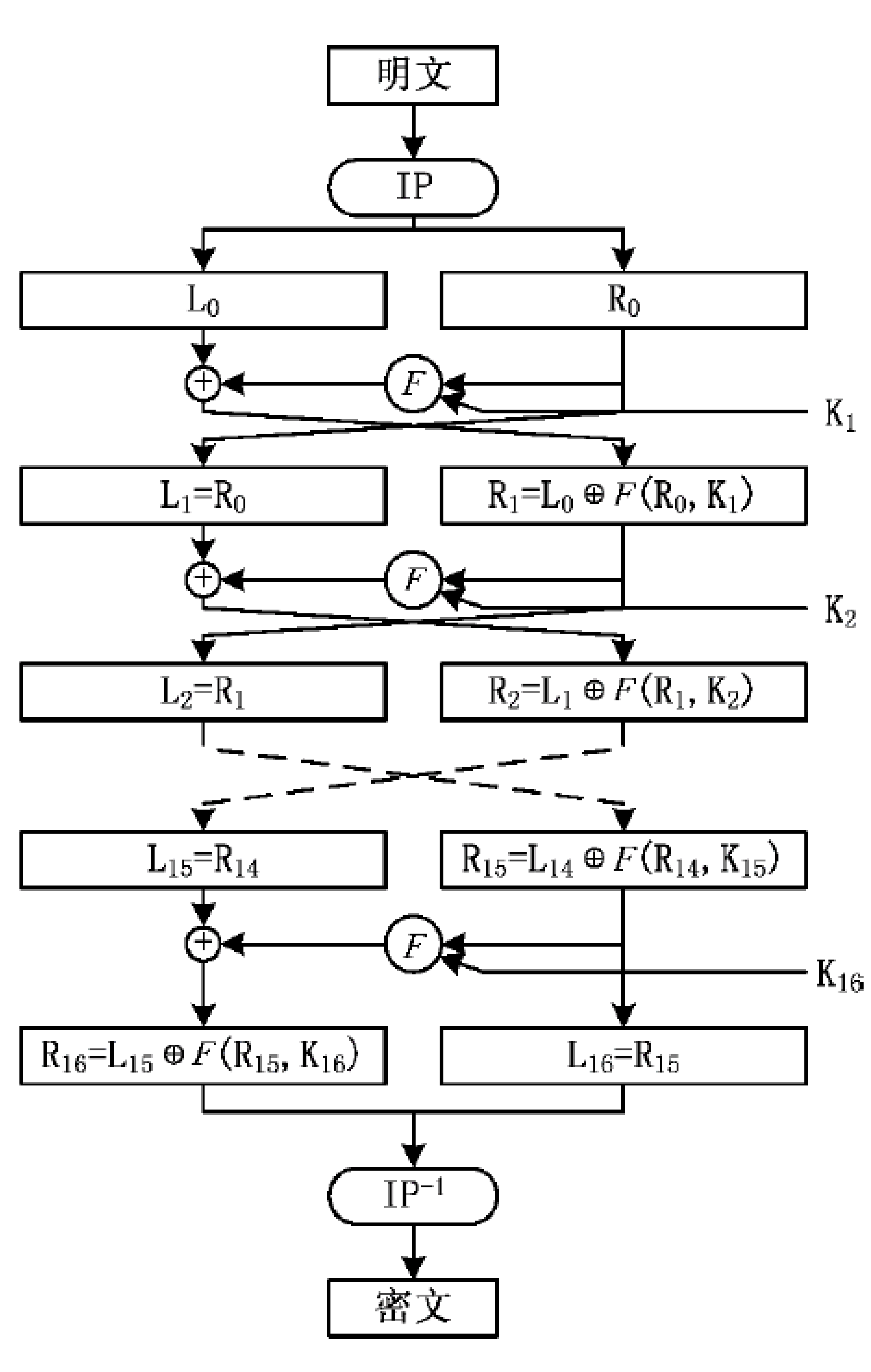


图 4-1　DES 算法的基本过程

右移来产生子密钥。解密子密钥循环右移的位数为：0,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1。这种解密模式是典型的 Feistel 密码结构的解密模式。

DES 算法的核心部分是基本过程中的第 3 步，第 3 步是关于每一轮的运算过程，该运算过程的具体步骤如图 4-2 所示。

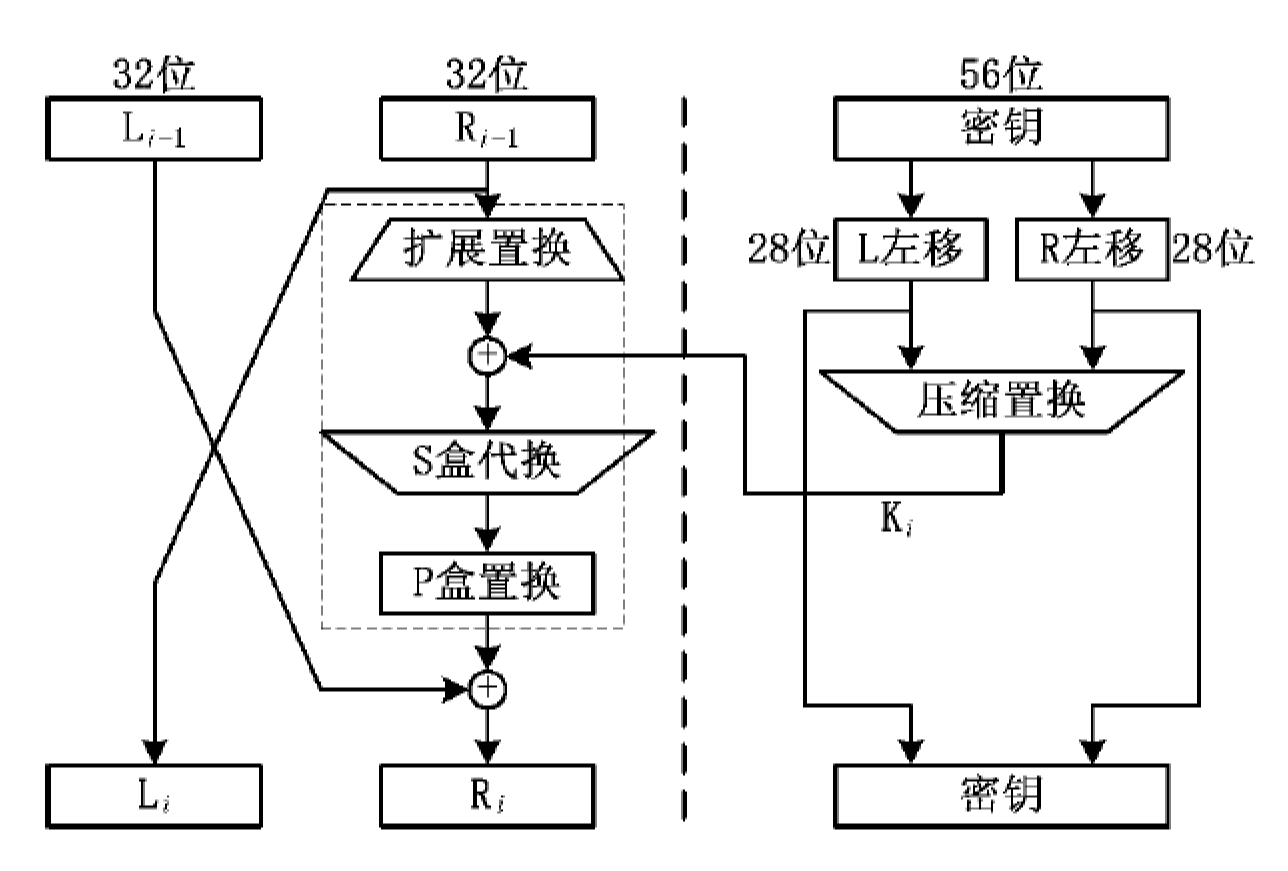


图 4-2 单轮 DES 加密过程

图 4-2 的左半部分为一轮的加密过程，右半部分为这一轮加密密钥的生成方法。每一轮的加密都采用不同的子密钥。

4.2 DES 密钥生成

DES 算法输入的密钥为 64 位，由于不考虑每个字节的第 8 位，那么 DES 算法的密钥就从 64 位减至 56 位，这个过程通过一置换操作来完成，置换方法见表 4-1。

表 4-1 DES 密钥初始置换

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

通过表 4-1 的置换操作去掉了每个字节的第 8 位，同时还对输入的密钥进行了混淆操作。在获得了 56 位的密钥之后，将 56 位的密钥分成两部分。根据轮数不同对左右两部分分别循环左移 1 位或 2 位。每轮移位的多少见表 4-2。

表 4-2 每轮循环左移的位数

轮	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
位数	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

在完成循环左移之后对获得的密钥再进行压缩置换，压缩置换进行了两部分的操作，一部分是进行压缩，将 56 位的密钥压缩为 48 位的密钥，另一部分是进行置换操作，其结果是实现了对密钥的混淆。

压缩置换的具体细节见表 4-3。

表 4-3 压缩置换

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

经过压缩置换后获得的密钥用于每一轮的加密。

解密密钥与加密密钥相同，使用的时候正好与加密密钥的顺序相反。

4.3 DES 算法加密过程

DES 加密算法的输入明文为 64 位的明文，其加密过程的第 1 步是对输入的明文进行 IP 置换(见图 4-1)，置换的方法见表 4-4。

表 4-4 IP 置换

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

经过 IP 置换得到混淆的明文，将 IP 置换后得到的数据分割成左右两部分，将右半部分的 32 位扩展到 48 位，这个操作产生了与密钥长度相同的数据，可以和密钥进行异或操作。扩展置换的具体方法见表 4-5。

表 4-5 扩展置换

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

在表 4-5 中，中间部分为输入的数据，左右两部分为扩展得到的数据，扩展置换的原理见图 4-3。

在经过扩展置换之后，获得的数据的位数与密钥的位数相同，均为 48 位。此时将扩展置换得到的数据与密钥进行“异或”运算，并将“异或”运算的结果进行 S 盒代换。

DES 算法的 S 盒共有 8 个，48 位的输入被分为 8 个 6 位的分组，每个分组对应一个 S 盒进行代换操作：分组 1 由第 1 个 S 盒进行操作，分组 2 由第 2 个 S 盒进行操作，以此类

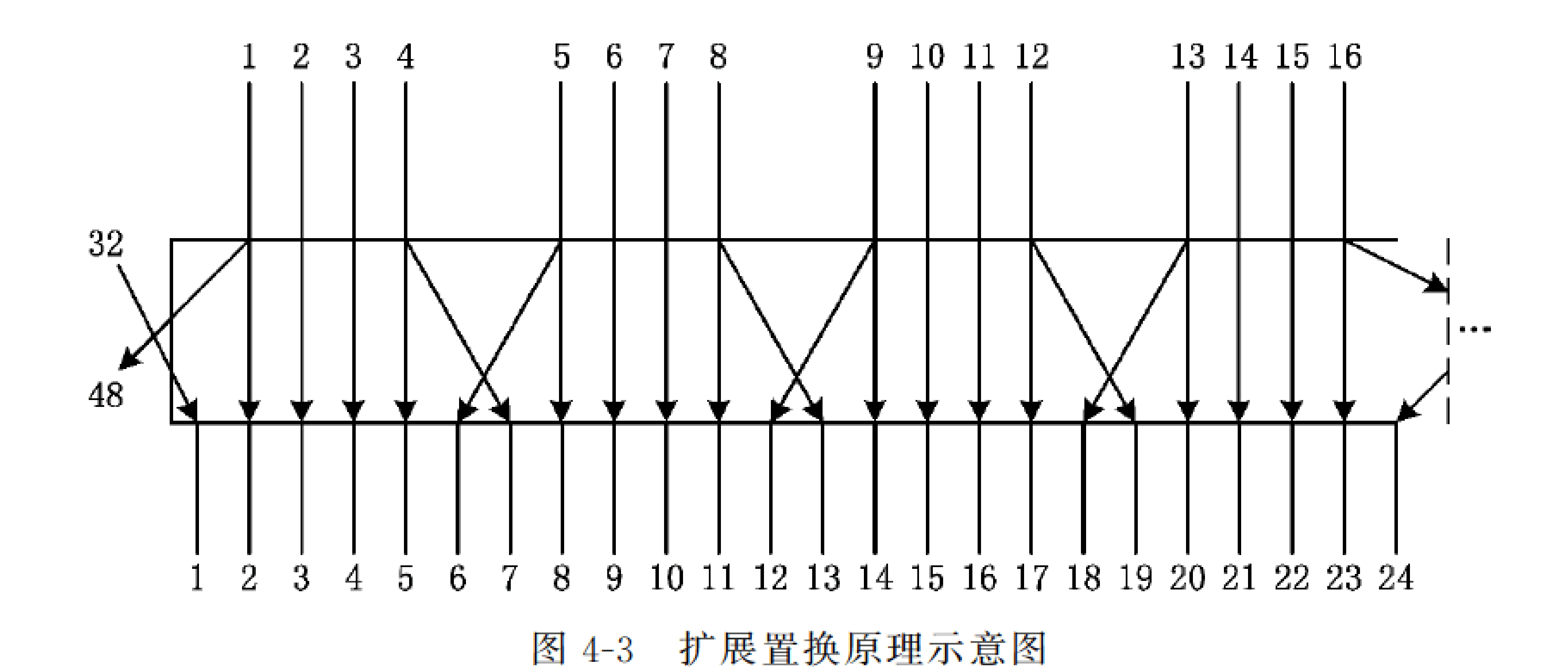


图 4-3 扩展置换原理示意图

推。每个 S 盒是 6 位输入,4 位输出,因此 48 位的输入数据经过 S 盒代换后输出 32 位数据。S 盒的代换基本原理见图 4-4。

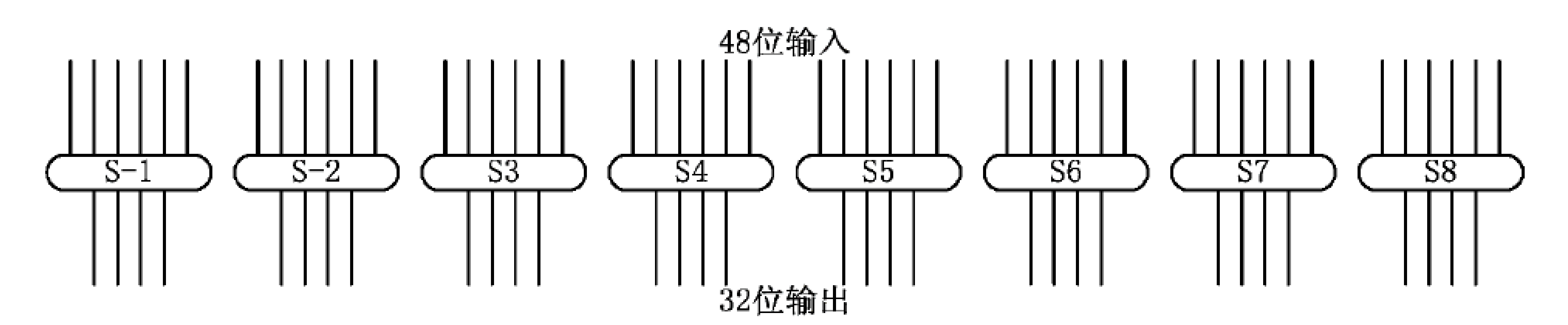


图 4-4 S 盒代换示意图

S 盒代换输出见表 4-6。

表 4-6 S 盒代换

S1	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S2	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S3	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S4	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S5	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

续表

S6	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S7	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S8	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	3
	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

DES 算法确定 S 盒项的方法与 S-DES 算法类似。假设 S 盒的输入是 $a_1, a_2, a_3, a_4, a_5, a_6$ 的 6 位输入，那么 a_1 和 a_6 组合构成一个 2 位数，从 0 到 3，确定 S 盒的行。从 a_2 到 a_5 构成一个 4 位数，从 0 到 15，确定 S 的列。

例如，S 盒的输入是 110101。第 1 位和第 6 位组合就得到 11，它对应着 S 盒的第 3 行（注：S 盒是从第 0 行、第 0 列开始计算的）。中间的 4 位为 1010，对应第 10 列。以 S1 盒为例计算输出，S1[3][10]=3，那么对应的输出则为 0110。

S 盒代换是 DES 算法最关键的一步，DES 算法的其他步骤的计算都是线性的，只有 S 盒是非线性运算，它对 DES 的安全性起到了关键作用。

S 盒代换后获得 32 位的输出，将 S 盒的输出进行 P 盒运算，P 盒运算实际上就是一种置换运算，P 盒的置换方法见表 4-7。

表 4-7　P 盒置换

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	23

将 P 盒置换后得到的结果与最初的 64 位分组的左半部分进行“异或”，然后按照图 4-2 所示的方法将左、右两部分进行交换，在进入下一轮的运算。

在第 16 轮运算结束时，左、右两半不再进行交换。将左、右两半合并在一起形成输出，然后进行末置换。末置换其实就是初始置换的逆过程。末置换的方法见表 4-8。

表 4-8　末置换

40	8	48	16	56	24	64	32	39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30	37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28	35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26	33	1	41	9	49	17	57	25

末置换的输出就是整个加密过程的输出。

解密过程除密钥的使用顺序不同以外，其他过程和加密过程完全相同。

4.4 DES 算法实现

DES 算法的实现过程中主要需要解决以下问题：

（1）用于密钥生成和加密过程中的各种置换运算。

（2）输入密钥、密钥生成、加密解密过程中的数据分割。

（3）E/P 扩展。

（4）S 盒运算。

（5）各种移位运算。

上述的各类需要解决的问题大多数与 S-DES 算法中的问题类似，只是比 S-DES 略为复杂一些。DES 算法中需要处理的各种数据以及相关的操作均封装在 DES 类中，DES 类的具体结构如图 4-5 所示。

DES	
- key	: unsigned long long
- plainText	: unsigned long long
- cipherText	: unsigned long long
- decipherText	: unsigned long long
- encKey[16]	: unsigned long long
- IP[64]	: static const int
- IPI[64]	: static const int
- keyIP[56]	: static const int
- encKeyRound[16]	: static const int
- CP[48]	: static const int
- EP[48]	: static const int
- SBox[32][16]	: static const int
- P[32]	: static const int
+ DES ()	
+ setKey (string k)	: void
+ setPlainText (string p)	: void
+ permutations (unsigned long long num, const int p[], int pmax, int n)	: unsigned long long
+ genEncKey ()	: void
+ SBoxes (unsigned long long num)	: unsigned long long
+ encryption() ()	: void
+ decryption() ()	: void
- keyLShift ()	: unsigned long long

图 4-5 DES 类的基本结构

DES 类声明的代码见程序清单 4-1。

程序清单 4-1

```
class DES
{
    public:
        DES();
        void setKey(string k);
        void setPlainText(string p);
        unsigned long long permutations
```

```
            (unsigned long long num,const int p[],int pmax,int n);
        void genEncKey();
        unsigned long long SBoxes(unsigned long long num);
        void encryption();
        void decryption();
        void showBinary(unsigned long long num);
        void showResult();
    private:
        unsigned long long keyLShift(unsigned long long k,int n);
        unsigned long long key;
        unsigned long long plainText;
        unsigned long long cipherText;
        unsigned long long decipherText;
        unsigned long long encKey[16];
        static const int IP[64];
        static const int IPI[64];
        static const int keyIP[56];
        static const int encKeyRound[16];
        static const int CP[48];
        static const int EP[48];
        static const int SBox[32][16];
        static const int P[32];
};
```

在程序清单 4-1 中各变量、常量、函数的功能如下：

- key——输入的密钥。
- cipherText——输入的明文。
- decipherText——解密密文得到的明文。
- encKey[16]——加密密钥，共 16 个。
- IP[64]——加密明文或解密密文时的初始置换数组。
- IPI[64]——加密明文或解密密文时的末置换数组。
- keyIP[56]——密钥置换数组，用于将密钥由 64 位转换为 56 位。
- encKeyRound[16]——用于生成子密钥时确定左移多少的数组。
- CP[48]——密钥压缩置换的数组，用于将密钥由 56 位压缩为 48 位。
- EP[48]——扩展置换数组，用于将 32 位的明文扩展为 48 位明文。
- SBox[32][16]——用于 S 盒运算的二维数组。
- P[32]——在经过 S 盒运算后的 P 置换数组。
- DES()——DES 类的构造函数，用于初始化数据。
- setKey()——用于设置初始密钥的函数。
- setPlainText()——用于设置明文的函数。
- permutations()——用于密钥生成、加密和解密等过程的各类置换的函数。
- genEncKey()——用于生成加密、解密用子密钥的函数。
- SBoxes()——用于 S 盒运算的函数。

- encryption()——用于加密的函数。
- decryption()——用于解密的函数。
- showBinary()——将数据以二进制形式显示的函数,用于检查数据计算过程。
- showResult()——用于显示解密、解密等结果的函数。

DES算法的密钥和加密的数据长度的最大位数为64位,因此在程序中利用了unsigned long long型数据,unsigned long long型数据的长度正好为64位,这样可以在一定程度上简化数据的处理。

本例中DES算法根据以下几个过程来完成:

(1) 初始化数据。

(2) 计算加密和解密用的子密钥。

(3) 对明文进行加密,对加密后的明文进行解密。

4.4.1 初始化数据

初始化数据由构造函数、获取密钥函数和获取明文函数来完成,同时还需要初始化各置换数组及S盒。构造函数用于初始化明文、密文、解密后的明文和密钥。构造函数的具体代码见程序清单4-2。

程序清单 4-2

```
DES::DES()
{
    key=0;                  //初始化密钥为0
    plainText=0;            //初始化明文为0
    cipherText=0;           //初始化密文为0
    decipherText=0;         //初始化解密明文为0
}
```

构造函数比较简单,只是将需要初始化的数据置为0。

各类常量数组的初始化见程序清单4-3。

程序清单 4-3

```
const int DES::IP[64]={
    58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
    62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
    57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
    61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7};
const int DES::IPI[64]={
    40,8,48,16,56,24,64,32,39,7,47,15,55,23,63,31,
    38,6,46,14,54,22,62,30,37,5,45,13,53,21,61,29,
    36,4,44,12,52,20,60,28,35,3,43,11,51,19,59,27,
    34,2,42,10,50,18,58,26,33,1,41,9,49,17,57,25};
const int DES::keyIP[56]={
    57,49,41,33,25,17,9,1,58,50,42,34,26,18,
    10,2,59,51,43,35,27,19,11,3,60,52,44,36,
```

```
        63,55,47,39,31,23,15,7,62,54,46,38,30,22,
        14,6,61,53,45,37,29,21,13,5,28,20,12,4};
const int DES::encKeyRound[16]={1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
const int DES::CP[48]={
        14,17,11,24,1,5,3,28,15,6,21,10,
        23,19,12,4,26,8,16,7,27,20,13,2,
        41,52,31,37,47,55,30,40,51,45,33,48,
        44,49,39,56,34,53,46,42,50,36,29,32};
const int DES::EP[48]={
        32,1,2,3,4,5,4,5,6,7,8,9,
        8,9,10,11,12,13,12,13,14,15,16,17,
        16,17,18,19,20,21,20,21,22,23,24,25,
        24,25,26,27,28,29,28,29,30,31,32,1};
const int DES::SBox[32][16]={
        14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,        //S1
        0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
        4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
        15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13,
        15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10,        //S2
        3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5,
        0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,
        13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9,
        10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8,        //S3
        13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1,
        13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,
        1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12,
        7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,        //S4
        13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9,
        10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,
        3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14,
        2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,        //S5
        14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6,
        4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,
        11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3,
        12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11,        //S6
        10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8,
        9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,
        4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13,
        4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,        //S7
        13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6,
        1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,
        6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12,
        13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,        //S8
        1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,
        7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,
```

```
        2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11};
const int DES::P[32]={
        16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,
        2,8,24,14,32,27,3,9,19,13,30,6,22,11,4,25};
```

常量中的 S 盒采用了二维数组进行处理，而不是采用三维数组，在加密运算过程中会对 S 的使用方法做进一步的说明。

设置密钥通过函数 setKey()来完成，setKey()函数的具体代码见程序清单 4-4。

程序清单 4-4

```
01 void DES::setKey(string k)
02 {
03         int i;
04         unsigned long long c;
05         for(i=0;i<8;i++)
06         {
07                 c=k[i];
08                 key=(c<<(7-i)*8)|key;
09     }
10 }
```

设置密钥函数的参数为字符串，设置密钥函数的功能为将包含 8 个字符的字符串转换为 unsigned long long 型数据，代码中的第 8 行利用移位的方法实现将输入按顺序放置到密钥对应的位置。注意事项为先将字符转换为 unsigned long long 型数据，否则字符移位后会出现丢失数据的问题。

在设置密钥过程中的具体移位方法见图 4-6。

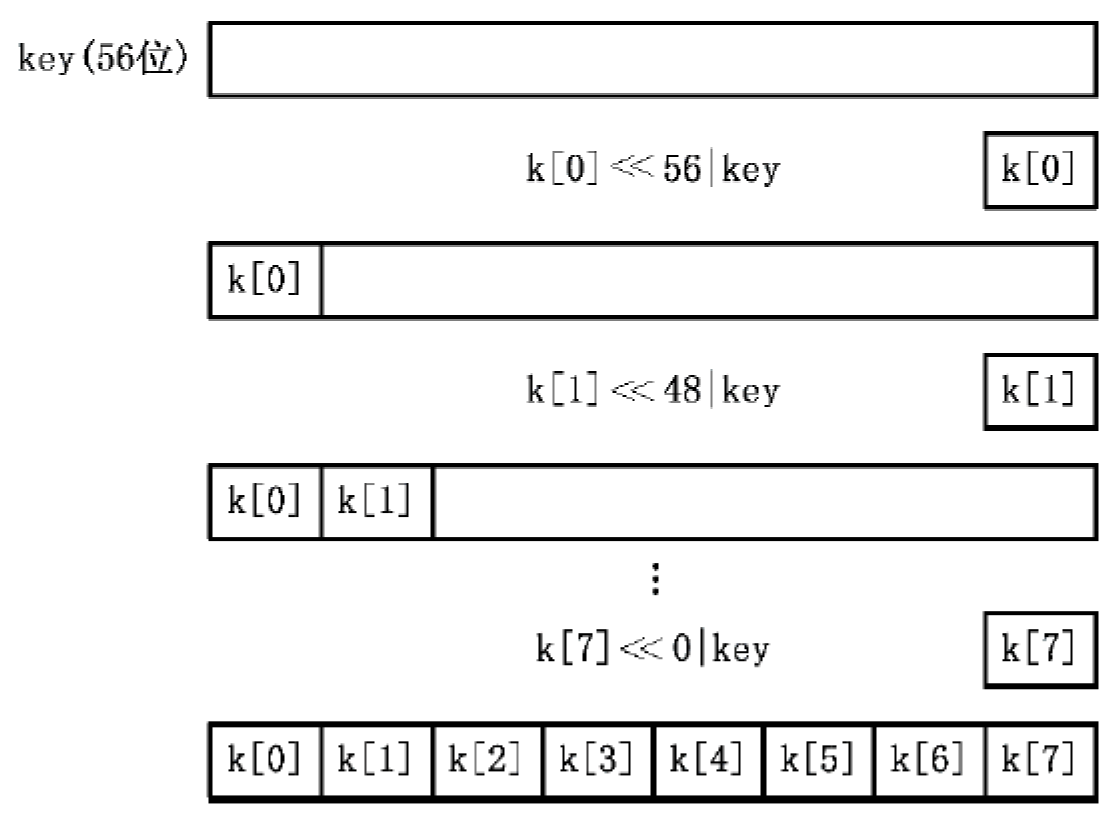


图 4-6 密钥设置过程中的移位计算方法

设置明文通过函数 setPlainText()来完成，setPlainText()函数的具体代码见程序清单 4-5。

程序清单 4-5

```
void DES::setPlainText(string p)
```

```
{
    int i;
    unsigned long long c;
    for(i=0;i<8;i++)
    {
        c=p[i];
        plainText=(c<<(7-i)*8)|plainText;
    }
}
```

设置明文的处理过程和注意事项与设置密钥的处理过程和注意事项完全一样。

4.4.2 生成子密钥

在 DES 算法中需要生成加密和解密用的子密钥。子密钥生成通过以下过程完成：

- 通过置换将 64 位密钥转换为 56 位密钥。
- 将密钥左右两半按照一定规则进行循环左移。
- 将循环左移得到的密钥进行压缩置换，获得 48 位的子密钥。

在程序中子密钥的生成通过函数 genEncKey()来完成，genEncKey()函数的代码见程序清单 4-6。

程序清单 4-6

```
01 void DES::genEncKey()
02 {
03     unsigned long long gKey;                          //临时计算的密钥
04     gKey=permutations(key,keyIP,64,56);               //密钥初始置换
05     int i;
06     for(i=0;i<16;i++)
07     {
08         gKey=keyLShift(gKey,encKeyRound[i]);
09         encKey[i]=permutations(gKey,CP,56,48);        //压缩置换
10     }
11 }
```

程序代码子密钥的生成分为以下几步：第 1 步是通过置换操作将 64 位的密钥置换为 56 位的密钥，这部分的工作由第 4 行代码完成，即调用 permutations()函数实现置换。第 2 步是通过代码第 6 行到第 8 行的过程来生成 16 个子密钥，其中分别调用了循环左移 keyLShift()函数和置换函数 permutations()。

循环左移函数 keyLShift()的代码见程序清单 4-7。

程序清单 4-7

```
unsigned long long DES::keyLShift(unsigned long long k,int n)
{
    unsigned long long tempKey=0;
    unsigned long long L,R;                        //密钥的左右两半
```

```
        L= (k&0xFFFFFFF0000000LL)>>28;
        R= k&0x0000000FFFFFFF;
        if(n==0)
        {
            tempKey= k;
        }
        if(n==1)
        {
            L= ((L&0x7FFFFFF)<<1)|((L>>27)&1);
            R= ((R&0x7FFFFFF)<<1)|((R>>27)&1);
            tempKey= (L<<28)|R;
        }
        if(n==2)
        {
            L= ((L&0x3FFFFFF)<<2)|((L>>26)&3);
            R= ((R&0x3FFFFFF)<<2)|((R>>26)&3);
            tempKey= (L<<28)|R;
        }
        return tempKey;
    }
```

移位函数的返回类型为 unsigned long long 型,函数的参数分别为需要移位的数据和循环左移的位数。由于在 DES 算法中只有针对 56 位的数据进行循环移位,因此该函数只适合对 56 位的输入进行操作。

该函数首先将输入分割成两部分,获得 L 和 R,以获得左半部分 L 为例来说明分割过程,其计算过程如下:

k	1011110110001110101000010100	1100011111111010000010100010
0xFFFFFFF0000000	1111111111111111111111111111	0000000000000000000000000000
k&0xFFFFFFF0000000	1011110110001110101000010100	0000000000000000000000000000
k&0xFFFFFFF0000000>>28	0000000000000000000000000000	1011110110001110101000010100

在移位过程中先执行了 k&0xFFFFFFF0000000LL,这是由于移位操作在不同的编译器中可能产生不同的结果,所以通过 & 运算将其他位的数据置为 0。以类似的方法可以获得右半部分 R。

循环左移也是利用了位运算的技巧,下面以获得的左半部分循环左移为例来说明。

L	1011110110001110101000010100
0x7FFFFFF	0111111111111111111111111111
L&0x7FFFFFF	0011110110001110101000010100
(L&0x7FFFFFF)<<1	0111101100011101010000101000
L>>27&1	0000000000000000000000000001
((L&0x7FFFFFF)<<1)|(L>>27&1)	0111101100011101010000101001

L&0x7FFFFFF 的作用是将 28 位数据的第 1 位置为 0。然后再移位，这样可以确保其他数据位的数据不变。其他部分的移位操作相同。

在获取密钥的函数中还用到了 permutations()，该函数的作用是实现置换，permutations()函数的代码见程序清单 4-8。

程序清单 4-8

```
unsigned long long DES::permutations
                    (unsigned long long num,const int p[],int pmax,int n)
{
    unsigned long long temp=0;
    int i;
    for(i=0;i<n;i++)
    {
        temp<<=1;         //temp 左移一位
        temp|=(num>>(pmax-p[i]))&1;
    }
    return temp;
}
```

permutations()函数的实现方法与 S-DES 算法中的 permutations()函数完全一样。

4.4.3　加密和解密

在获得子密钥之后就可以对明文进行加密，或对密文进行解密。加密或解密的过程由以下步骤完成：

- 将输入的明文进行一初始置换。
- 将置换后的明文进行分组，分成左右各一半，各为 32 位数据。
- 进行 16 轮完全相同的 F 运算。
- 将 F 运算获得的结果左右结合，然后再进行一个末置换得到密文。

在加密和解密过程中，核心部分是 F 运算，F 运算的过程为：

- 将输入的右半部分的 32 位数据进行扩展置换，得到 48 位数据。
- 将 48 位的数据与密钥进行“异或”运算。
- 将“异或”运算的结果进行 S 盒运算，替代为 32 位的数据。
- 将 S 盒运算的结果再进行一次置换运算。

在经过 F 运算后得到 32 位的输出，将这 32 位的输出与左半部分进行“异或”运算，得到新一轮的右半部分数据，而原来的右半部分数据成为新一轮运算的左半部分数据。

加密程序通过函数 encryption()完成，函数 encryption()的代码见程序清单 4-9。

程序清单 4-9

```
void DES::encryption()
{
    unsigned long long temp=permutations(plainText,IP,64,64);
    int i,j;
    unsigned long long L,R,tempR;
```

```
06        L=(temp&0xFFFFFFFF00000000LL)>>32;
07        R=(temp&0x00000000FFFFFFFFLL);
08        tempR=R;
09        for(i=0;i<16;i++)
10        {
11            tempR=permutations(R,EP,32,48);
12            tempR=tempR^encKey[i];
13            tempR=SBoxes(tempR);
14            tempR=permutations(tempR,P,32,32);
15            tempR=tempR^L;
16            L=R;
17            R=tempR;
18        }
19        temp=(R<<32)|L;
20        temp=permutations(temp,IPI,64,64);
21        cipherText=temp;
22    }
```

在程序清单 4-9 中,第 3 行代码是实现明文的初始置换。第 6 行和第 7 行代码是将输入分割成左右两部分,各为 32 位。第 9 行到第 18 行代码是实现 16 轮的加密运算,其中第 11 行代码是实现扩展置换,第 12 行代码是实现与子密钥的"异或"运算,第 13 行是 S 盒运算,第 14 行是置换运算,第 15 行是将置换的结果与左半部分进行"异或",然后进行左右两部分的交换。经过 16 轮运算之后,第 19 行将左右两部分合并,再由第 20 行代码进行末置换,最后得到加密的密文。

在 encryption()函数中调用了 SBoxes()函数对输入进行 S 盒运算,该函数的代码见程序清单 4-10。

程序清单 4-10

```
unsigned long long DES::SBoxes(unsigned long long num)
{
    int i;
    unsigned long long temp;
    unsigned long long result=0;
    for(i=0;i<8;i++)
    {
        temp=(num>>((7-i)*6))&0x3F;
        int x=((temp>>4)&0x2)|(temp&0x1)+i*4;
        int y=(temp>>1)&0xF;
        temp=SBox[x][y];
        temp=temp<<((7-i)*4);
        result=result|temp;
    }
    return result;
}
```

S 盒运算函数实现的将输入的 48 位数据每次读取 6 位，然后将其他位数置为 0。代码行的第 8 行实现了这一功能。第 9 行和第 10 行是获取 S 盒的行列值，行是通过获得第 1 位和第 6 位的数据来实现，其中 i * 4 是用来确定读取的是第几个 S 盒，每个 S 盒占 4 行数据。因为输入 6 位，输出 4 位，因此输入 48 位输出 32 位，所以在将得到的输出移到原来位置时按比例减少位数，第 12 行实现这一功能。然后与应该输出的数据进行“或”运算，得到相应的输出。经过 8 轮运算后得到 S 盒运算结果。

解密过程通过 decryption()函数来实现，decryption()的代码见程序清单 4-11。

程序清单 4-11

```
void DES::decryption()
{
    unsigned long long temp=permutations(cipherText,IP,64,64);
    int i,j;
    unsigned long long L,R,tempR;
    L=(temp&0xFFFFFFFF00000000LL)>>32;
    R=(temp&0x00000000FFFFFFFFLL);
    tempR=R;
    for(i=0;i<16;i++)
    {
        tempR=permutations(R,EP,32,48);
        tempR=tempR^encKey[15-i];
        tempR=SBoxes(tempR);
        tempR=permutations(tempR,P,32,32);
        tempR=tempR^L;
        L=R;
        R=tempR;
    }
    temp=(R<<32)|L;
    temp=permutations(temp,IPI,64,64);
    decipherText=temp;
}
```

解密函数与加密函数的实现方法基本相同，不同的地方在于密钥使用的顺序正好与加密过程相反。具体各部分实现的功能可以参考加密函数。

在 DES 类中还有一个函数 showBinary()，这个函数可以用来检查数据的二进制显示，在程序中使用了 vector 来临时存储数据，具体代码见程序清单 4-12。

程序清单 4-12

```
void DES::showBinary(unsigned long long num)
{
    vector<int>v;
    do
    {
        v.push_back(num%2);
```

```
07            num=(num-num%2)/2;
08        }while(num!=0);
09        for(int i=(v.size()-1);i>=0;i--)
10        {
11            cout<<v[i];
12        }
13        cout<<endl;
14    }
```

若在程序中需要检查数据的二进制显示形式，可以直接调用该函数。将数据转换为二进制的过程由代码第 4 行到第 8 行实现，通过判断数据的奇偶性来确定存储到向量的值，如果是奇数，则往向量中存储“1”，如果是偶数，则往向量中存储“0”，直到数据处理完毕。最后通过第 9 行到第 12 行的代码将二进制数据输出。

本程序的主要目的是解决 DES 算法的基本原理的实现过程，因此在主程序中进行加密解密的过程需注意先后顺序，通常过程为：

- 设置密钥；
- 设置明文；
- 计算子密钥；
- 加密；
- 解密。

例如，可以通过程序清单 4-13 实现此过程。

程序清单 4-13

```
int main()
{
    DES des;
    des.setKey("!2bcde<~ ");
    des.setPlainText("ABs5EFGH");
    des.genEncKey();
    des.encryption();
    des.decryption();
    des.showResult();
    return 0;
}
```

程序运行结果如下：

```
key=!2bcde<~
plainText=ABs5EFGH
cipherText=祩檽▸↔0
decipherText=ABs5EFGH
```

主函数中使用了 DES 类中的 showResult()函数，showResult()函数的作用是显示整个加密和解密的计算结果，该函数的详细代码见程序清单 4-14。

程序清单 4-14

```
void DES::showResult()
```

```
{
    int i;
    cout<<"key=";
    for(i=0;i<8;i++)
    {
        cout<<(char)((key>>(7-i)*8)&0xFF);
    }
    cout<<endl;
    cout<<"plainText=";
    for(i=0;i<8;i++)
    {
        cout<<(char)((plainText>>(7-i)*8)&0xFF);
    }
    cout<<endl;
    cout<<"cipherText=";
    for(i=0;i<8;i++)
    {
        cout<<(char)((cipherText>>(7-i)*8)&0xFF);
    }
    cout<<endl;
    cout<<"decipherText=";
    for(i=0;i<8;i++)
    {
        cout<<(char)((decipherText>>(7-i)*8)&0xFF);
    }
    cout<<endl;
}
```

showResult()函数的作用仅为输出计算结果，在显示结果的过程中，需要将 unsigned long long 型数据转换为 char 型数据，转换的方法是从高位开始，每次读取 8 位数据，然后向右移位，将该 8 位数据置于低位，再与 0xFF 进行"&"运算，最后，强制转化为 char 型数据并输出。输入的密钥、明文，以及加密后的结果和解密后的结果均采用同样的方法处理。该处理过程也可以写成一个单独的函数，函数的参数是 unsigned long long 型数据，函数的作用是将参数转化为 char 型数据并输出。

4.5 DES 算法的变种

DES 算法的安全性极大依赖于密钥的长度和 S 盒的设计，由于 DES 算法的密钥本质上是 56 位的密钥，密钥长度相对较短。由于计算机的运算速度自 DES 算法应用以来有了极大的提高，同时对 DES 算法的破解研究也有所发展，DES 算法的安全性受到了很大的挑战。因此，为保证数据的安全性，对 DES 算法进行了多种改进。这些改进包括多重 DES、独立子密钥的 DES、更换 S 盒的 DES 等。

4.5.1 三重 DES 算法

三重 DES 算法是 DES 算法的变种中最重要的一种，三重 DES 算法的基本原理如图 4-7 所示。

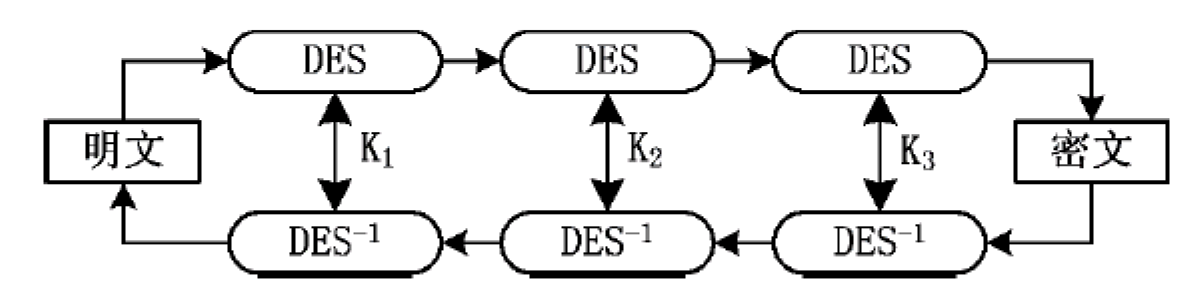


图 4-7 三重 DES 加密原理图

三重 DES 算法也分成两种，一种是双密钥加密模式，一种是三密钥的加密模式，双密钥的加密模式可以表示为

$$\begin{aligned} C &= E_{k1}(D_{k2}(E_{k1}(P))), \\ P &= D_{k1}(E_{k2}(D_{k1}(P)))。 \end{aligned} \tag{4-1}$$

双密钥加密模式的第 1 个密钥和第 3 个密钥是相同的，加密过程是：首先使用第 1 个密钥进行加密，然后使用第 2 个密钥进行解密，最后使用第 1 个密钥进行加密。解密过程是：首先使用第 1 个密钥进行解密，然后使用第 2 个密钥进行加密，最后使用第 1 个密钥进行解密。

使用三密钥的加密模式可以表示为

$$\begin{aligned} C &= E_{k3}(D_{k2}(E_{k1}(P))), \\ P &= D_{k1}(E_{k2}(D_{k3}(P)))。 \end{aligned} \tag{4-2}$$

三密钥加密模式的三个密钥都不相同，加密过程是：首先使用第 1 个密钥进行加密，然后使用第 2 个密钥进行解密，最后使用第 3 个密钥进行加密。解密过程是：首先使用第 3 个密钥进行解密，然后使用第 2 个密钥进行加密，最后使用第 1 个密钥进行解密。解密过程密钥的使用顺序正好与加密过程的密钥使用顺序相反。

4.5.2 独立子密钥的 DES 算法

除了多重 DES 算法外，还有一种变种的 DES 算法，这种算法在每一轮的 DES 加密过程中均采用了不同的密钥，这样使得密钥的长度为 768 位，采用这种算法对穷举攻击能够有效预防，极大增强了穷举攻击的难度。但独立子密钥的加密方法对选择明文攻击几乎没有效果，不能使 DES 算法变得更加安全。

三重密钥的 DES 算法和独立子密钥的 DES 算法的程序实现方法与 DES 算法的实现方法基本相同。

4.6 习题与实践题

4.6.1 习题

1. 请简要说明 DES 算法的解密和解密的基本过程。
2. 请简要说明 DES 算法中扩展置换的基本原理，并绘出扩展置换过程的示意图。

3. 试说明 DES 算法中 S 盒运算的基本方法，并给出示例说明 S 盒的工作原理。

4. 三重 DES 算法的基本工作模式有哪几种？并说明各三重 DES 算法的加密、解密过程，同时用计算方法说明。

5. 试比较 DES 算法和三重 DES 算法的安全性，并说明具体原因。

4.6.2 实践题

1. 参考 4.4 节中给出的 DES 算法的实现方法，实现三重 DES 加密算法，使用的加密和解密算法为

$$C = E_{k1}(D_{k2}(E_{k1}(P))),$$
$$P = D_{k1}(E_{k2}(D_{k1}(P)))。$$

待加密和解密的消息为 64 位(可以使用 8 个字符)，试完成该算法。

2. 参考 4.4 节中给出的 DES 算法的实现方法，实现三重 DES 加密算法，使用的加密和解密算法为

$$C = E_{k3}(D_{k2}(E_{k1}(P))),$$
$$P = D_{k1}(E_{k2}(D_{k3}(P)))。$$

待加密和解密的消息为 64 位(可以使用 8 个字符)，试完成该算法。

(注：可以选择一题作为实践练习。)

第 5 章

AES 算 法

AES(Advanced Encryption Standard)高级数据加密标准是美国国家标准技术局于2001年公布的新的加密标准,其目的是逐步取代DES算法,是目前被广泛使用的标准。在密码学中又称为Rijndael加密法,该算法为比利时密码学家Joan Daemen和Vincent Rijmen所设计,结合两位作者的名字,以Rijndael为名投稿高级加密标准的甄选流程,因此该算法又被称为Rijndael加密法。

5.1 置换-组合结构

DES加密算法使用的是Feistel密码结构,AES算法则采用了一种新型的密码结构,这个结构称为置换-组合(代换)结构(substitution-permutation network)。置换-组合结构主要用在分组加密中,在加密过程中进行一系列置换和组合操作。置换-组合结构的加密基本原理见图5-1。

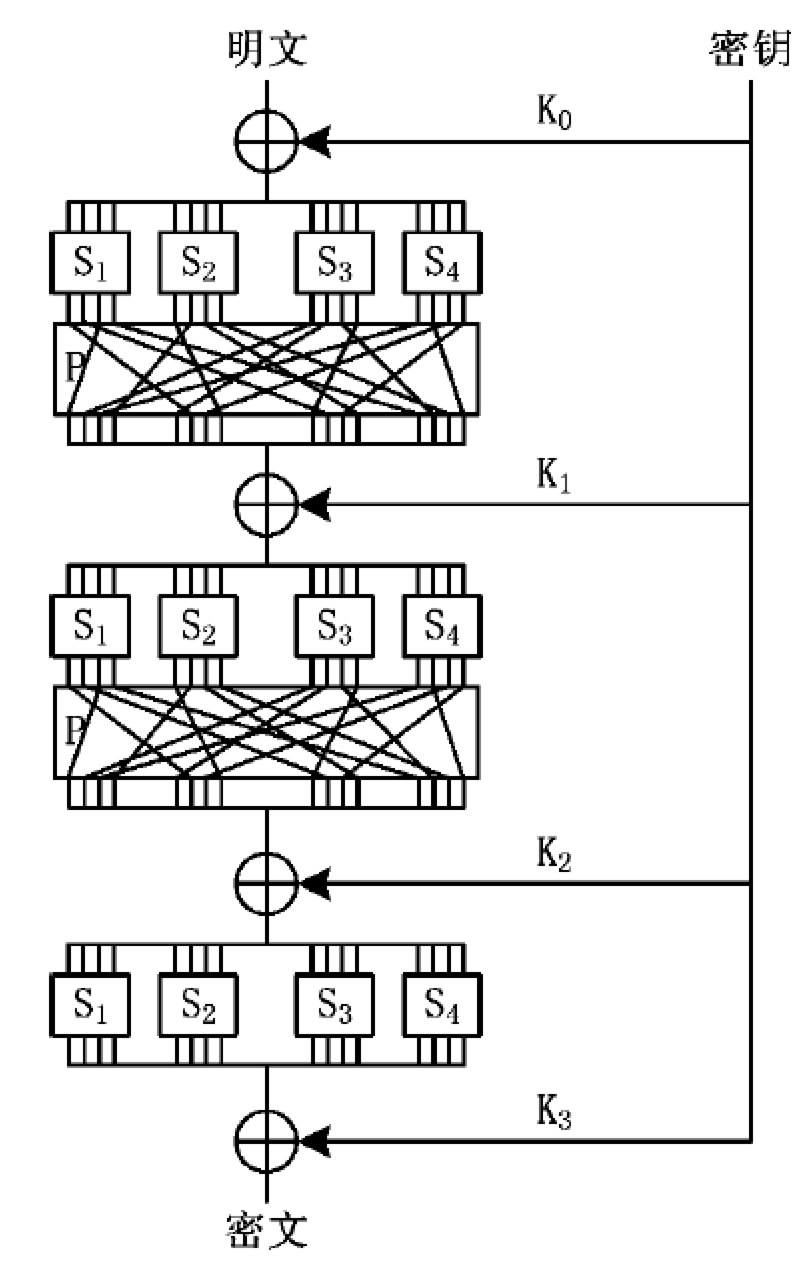


图5-1 置换-组合基本结构

置换-组合结构加密的基本输入是明文和密钥,然后经过数轮S盒运算和P盒运算得到最终密文。S盒和P盒的运算是按位进行运算的,这样在实现过程中将有较高的运算速度。密钥与输入进行的是按位"异或",同样具有很高的运算速度。

在加密过程中使用的密钥又称为轮密钥,轮密钥一般由输入密钥派生出来。在有些设计中将S盒与轮密钥关联,即S盒依赖于轮密钥。

置换-组合结构的解密过程是加密过程的逆过程,即S盒、P盒和轮密钥的使用顺序与加密过程正好相反。

S盒的运算方法是:将输入分为与S盒数目相同的若干组,每组与每个相应的S盒对应以确保S盒的可逆性。S盒的输入和输出长度相同(与DES算法不同,DES算法是6位输入,4位输出),通常S盒的输入和输出都是4位。S盒的运算是非线性运算。

P盒运算是针对S盒所有的输出进行的,将从S盒得到的输入经过P盒的置换或组合

操作,然后输出到下一轮 S 盒运算。理想的 P 盒是尽可能置换输入数据的每一位。

轮密钥运算则是进行简单的“异或”运算。

5.2 AES 算法原理

AES 算法的输入和输出是 128 位数据,密钥长度可以是 128 位、192 位或 256 位。从严格意义上来说,AES 算法不是完全的 Rijndael 加密法,Rijndael 加密法的密钥和分组长度是 32 位的整数倍,以 128 位为下限,256 位为上限。AES 密钥的生成方案是由 Rijndael 密钥生成方案提供。

AES 算法具有以下特点:

(1) 对所有已知的攻击具有免疫性。

(2) 在各种平台上,具有代码紧凑、运行速度快的优点。

(3) 设计简单。

AES 算法的基本流程见图 5-2。

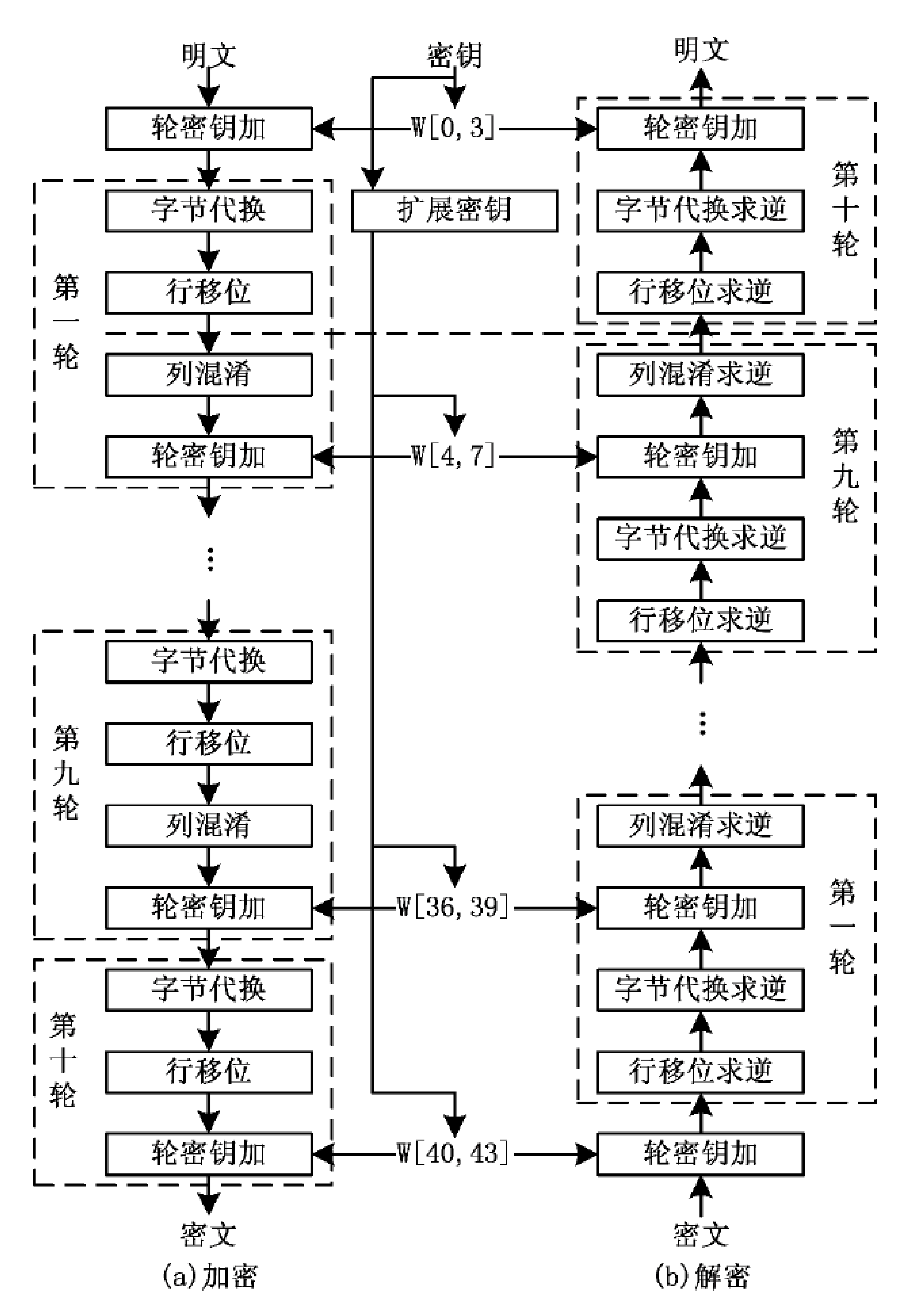


图 5-2 AES 算法加密解密过程示意图

图 5-2(a)为加密过程,图 5-2(b)为解密过程。解密过程是加密过程的逆过程。

在 AES 算法中加密的轮数是根据密钥长度和分组明文的长度来决定的。假设 N_b 表示数据分组长度/32,N_k 表示密钥分组长度/32,N_r 表示轮数,那么三者之间的关系见表 5-1。

表 5-1 轮数、密钥长度和明文分组长度关系

N_r	$N_b=4$	$N_b=8$	$N_b=8$
$N_k=4$	10	12	14
$N_k=6$	12	12	14
$N_k=8$	14	14	14

在 AES 算法中所有的运算都是针对字节的,因此可以将数据分组表示成以字节为单位的数组。加密过程是在一个 4×4 字节的矩阵上进行运算的,这个矩阵称为“state”(状态矩阵),其初值就是一个明文的分组。

AES 算法中的每一轮加密包含 4 个主要的步骤:轮密钥加、字节代换、行移位和列混淆。各部分的功能和算法如下。

(1) 轮密钥加——将输入状态矩阵与轮密钥矩阵进行“异或”(⊕)运算,轮密钥由密钥通过相应变换获得。轮密钥加运算的基本原理见图 5-3。

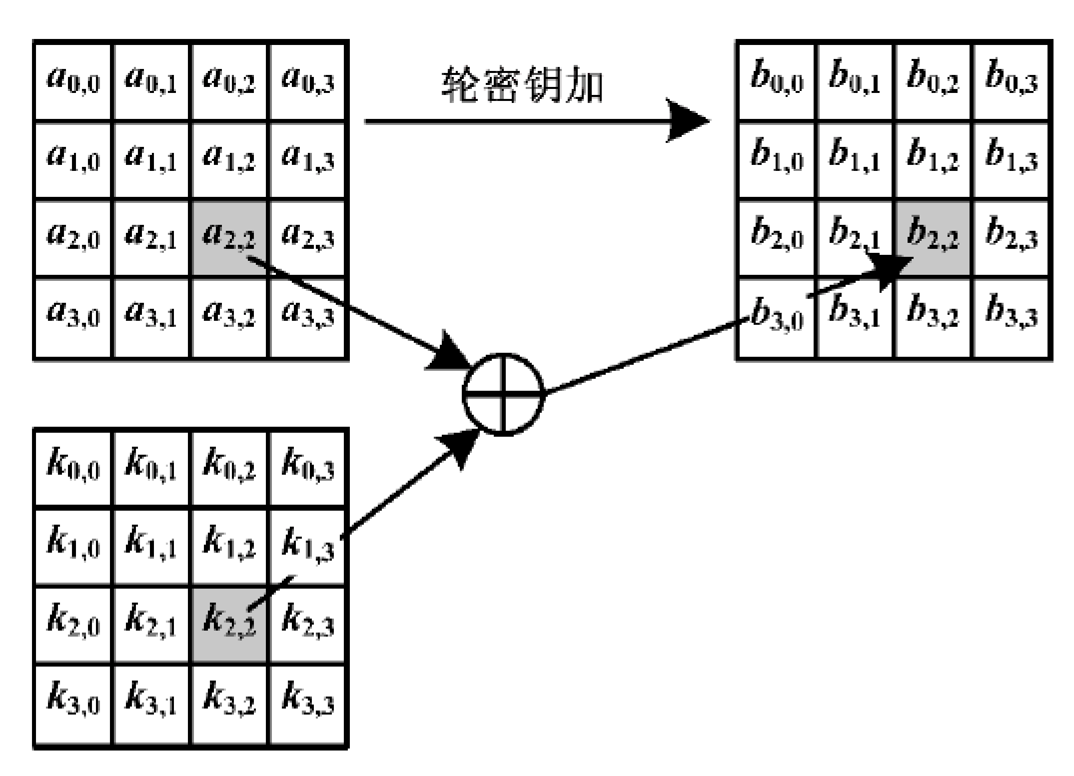


图 5-3 轮密钥加基本原理

解密过程的轮密钥加与加密过程的轮密钥加过程相同。

(2) 字节代换——将输入的状态矩阵与 S 盒代换表进行的非线性变换,其变换的基本原理见图 5-4。

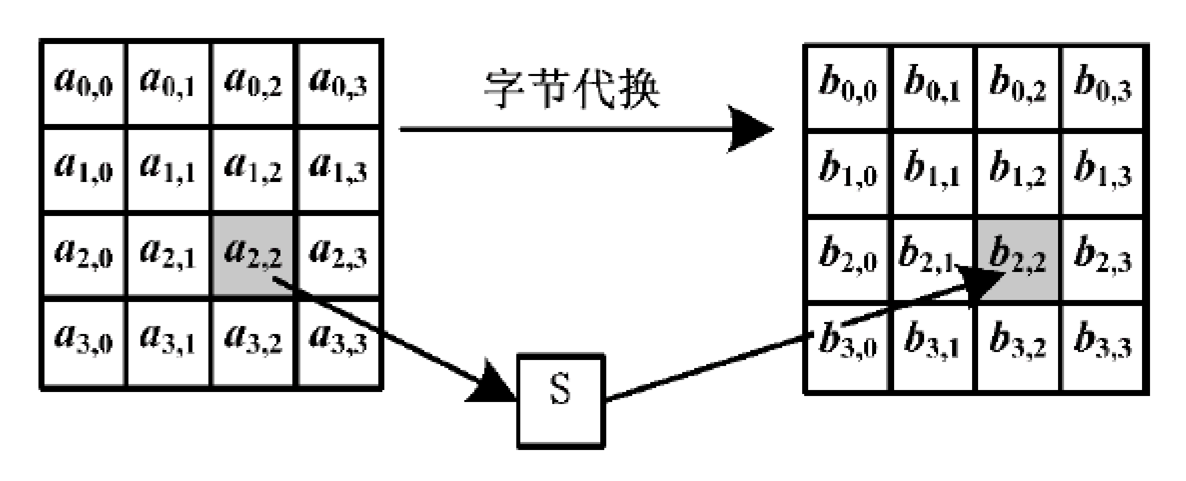


图 5-4 字节代换示意图

AES 算法中的 S 盒代换表是一个 16×16 的矩阵,它的构造方法如下:

(1) 逐行按升序排序的字节值初始化 S 盒。第 1 行的值是{00,01,02,…,0F},第 2 行

的值是{10,11,12,…,1F},以此类推,在第 x 行和第 y 列的值是{xy}。

(2) 把 S 盒中的每个字节映射为它在有限域 GF(2^8)中的逆,{00}被映射为自身{00}。

(3) 把 S 盒的每个字节记成($b_7,b_6,b_5,b_4,b_3,b_2,b_1,b_0$),对 S 盒中的每个字节的每位做如下变换:

$$b'_i = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i, \quad i = 0,1,\cdots,7 \tag{5-1}$$

在式(5-1)中,c_i 是指 c 的第 i 位,c 的值为 0x63,即($c_7c_6c_5c_4c_3c_2c_1c_0$)=(01100011)。式(5-1)的计算过程也可以用如下方式描述:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{5-2}$$

在具体的使用过程中可以使用计算得到的 S 盒,S 盒代换表的具体取值见表 5-2。

表 5-2 S 盒代换表

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	D8
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

S 盒的 x 和 y 的取值分别对应输入的高 4 位和低 4 位。

例如：假设 $a_{2,2}=\text{AB}$，那么 x 取输入 $a_{2,2}$ 的高 4 位，得 $x=\text{A}$，y 取输入 $a_{2,2}$ 的低 4 位，得 $y=\text{B}$，对应 S 盒得到的输出为 62。

在解密过程中包含了字节代换求逆的过程，在字节代换求逆的过程中使用了逆 S 盒，逆 S 盒的计算过程利用了式(5-2)的逆变换，该逆变换是由 $\text{GF}(2^8)$ 上的逆变换得到的。该逆变换为

$$b'_i = b_{(i+2)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus d_i, \quad i=0,1,\cdots,7 \tag{5-3}$$

其中 $d=\{05\}$ 或 00000101。式(5-3)的计算过程也可以用如下方式来描述：

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5-4}$$

通过式(5-4)计算得到的逆 S 盒置换表见表 5-3。

表 5-3 逆 S 盒代换

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	23	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	A4	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

逆 S 盒的 x 和 y 的取值也是分别对应输入的高 4 位和低 4 位。

(4) 行移位变换——行移位变换是对输入的状态矩阵进行变换。

行移位变换的基本过程是：状态矩阵的第 1 行保持不变，第 2 行循环左移 1 个字节，第 3 行循环左移 2 个字节，第 4 行循环左移 3 个字节。行移位变换的基本过程如图 5-5 所示。

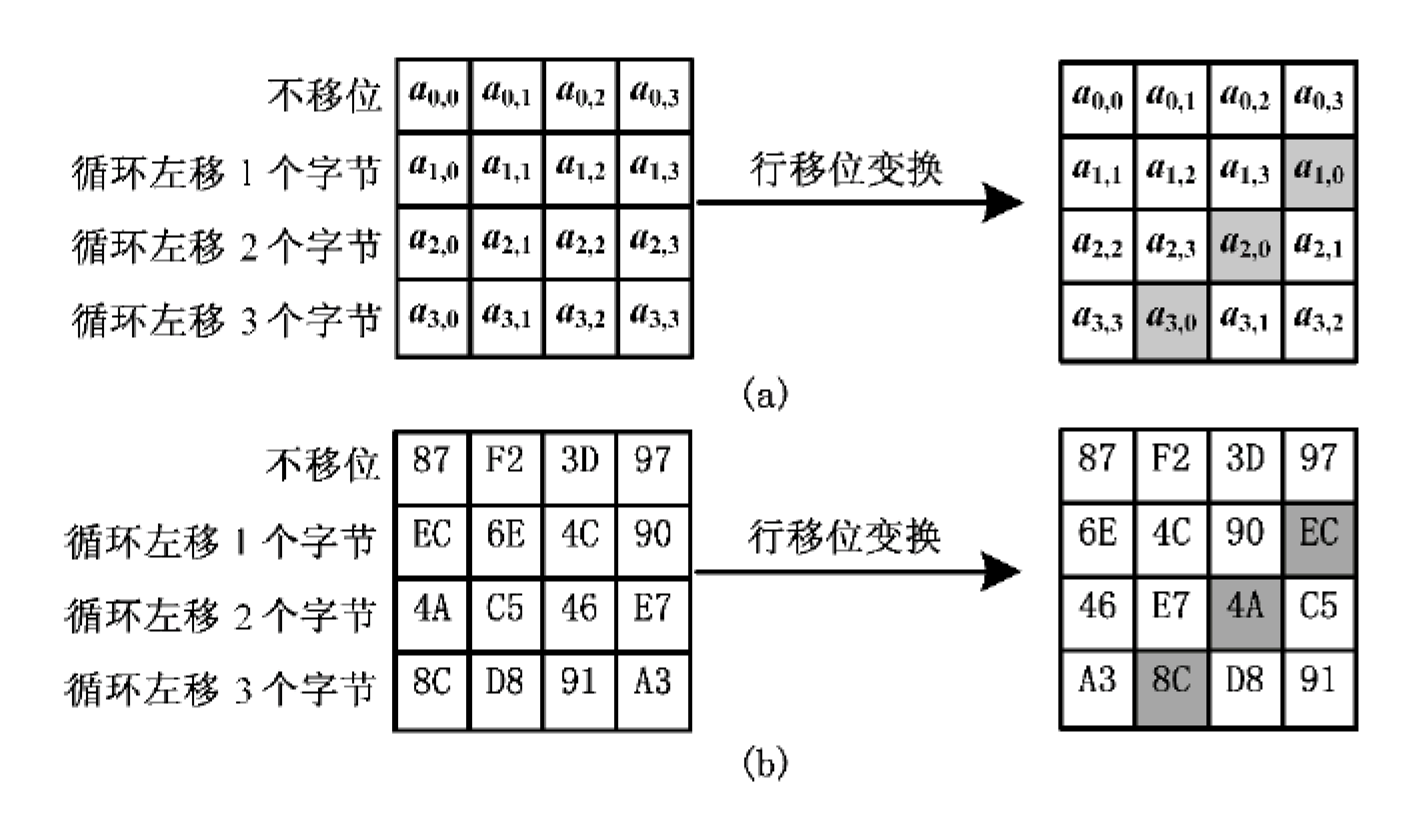


图 5-5　行移位变换示意图

图 5-5 中的(a)图为行移位变换的基本原理，(b)图为行移位变换的示例。

解密过程的行移位变换求逆，与加密过程的行移位变换过程正好相反，第 1 行也是不进行变换，第 2 行循环右移 1 个字节，第 3 行循环右移 2 个字节，第 4 行循环右移 3 个字节。行移位变换求逆的过程如图 5-6 所示。

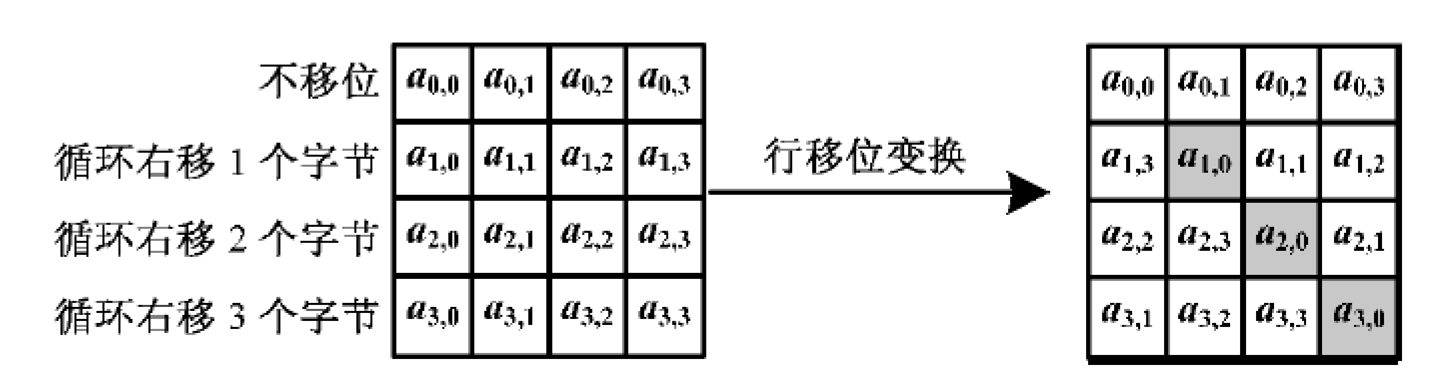


图 5-6　行移位变换求逆示意图

(5) 列混淆变换——是输入状态矩阵每一列的 4 个字节通过线性变换互相结合，列混淆的基本原理见图 5-7。

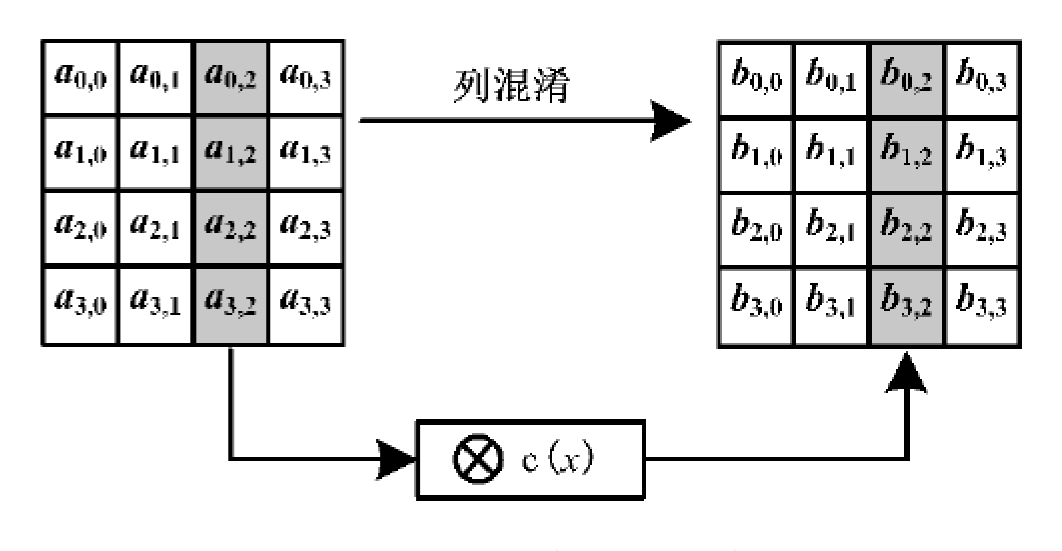


图 5-7　列混淆过程示意图

在运算过程中，每一列的 4 个元素分别当做 1，x，x^2 和 x^3 系数，合并即为 GF(2^8)中的一个多项式，将此多项式和一个固定的多项式 $c(x)=3x^3+x^2+x+2$ 在模 x^4+1 下相乘，这个运算过程可以表示为

$$\begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \tag{5-5}$$

乘积矩阵中的每个元素均是一行和一列中对应元素的乘积之和，乘法和加法都是定义在 $GF(2^8)$上的。状态矩阵的第 j 列的列混淆变换可以表示为

$$\begin{aligned} b_{0,j} &= (2 \cdot a_{0,j}) \oplus (3 \cdot a_{1,j}) \oplus a_{2,j} \oplus a_{3,j} \\ b_{1,j} &= a_{0,j} \oplus (2 \cdot a_{1,j}) \oplus (3 \cdot a_{2,j}) \oplus a_{3,j} \\ b_{2,j} &= a_{0,j} \oplus a_{1,j} \oplus (2 \cdot a_{2,j}) \oplus (3 \cdot a_{3,j}) \\ b_{3,j} &= (3 \cdot a_{0,j}) \oplus a_{1,j} \oplus a_{2,j} \oplus (2 \cdot a_{3,j}) \end{aligned} \tag{5-6}$$

在解密过程中使用了列混淆求逆，由于固定多项式与 x^4+1 互素，因此在解密过程中的运算为

$$\begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \tag{5-7}$$

其中

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5-8}$$

5.3 AES 密钥生成

AES 密钥生成的方法采用的是 Rijndael 密钥生成方法，通过密钥扩展来生成每轮运算所需要的子密钥。以密钥长度为 128 位为例，AES 密钥扩展算法的输入值是 4 字(16 字节，一字为 4 个字节)密钥，通过密钥扩展输出一个 44 字(156 字节)的一维线性数组，为初始轮密钥加和算法中的其他 10 轮运算提供每轮 4 字(16 字节)的子密钥，密钥扩展的过程可以通过程序清单 5-1 的伪码来描述。

程序清单 5-1

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp
    i=0
    while(i<Nk)
        w[i]=word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i=i+1
    end while
    i=Nk
```

```
    while(i<Nb*(Nr+1))
        temp=w[i-1];
        if(i mod Nk=0)
            temp=SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if(Nk>6 and i mod Nk=4)
            temp=SubWord(temp)
        end if
        w[i]=w[i-Nk] xor temp
        i=i+1
    end while
end
```

例如，如果密钥长度为 128 位，那么 N_k 为 128/32＝4。如果明文分组长度为 128 位，那么 N_b＝128/32＝4。通过表 5-1 可以得到 N_r＝10。

在程序清单 5-1 中还包括了 SubWord、RotWord 和 Rcon 三个运算，这三个运算过程分别是：

SubWord——利用 S 盒对输入的每个字节进行字节代换，S 盒见表 5-2。

RotWord——功能是使输入的 4 个字节循环左移 1 个字节。即输入$[a_0, a_1, a_2, a_3]$，变换为$[a_1, a_2, a_3, a_0]$。

Rcon——轮常量，将经过 RotWord 变换和 SubWord 变换后的数据与轮常量 Rcon[]相“异或”。

轮常量是一个字(4 个字节)，这个字的右边 3 个字节总为 0。因此与 Rcon 中的一个字相异或，其结果只是与该字最左边的那个字节相异或，通常轮常量只用其最左边的一个字节，Rcon[j]＝(RC[j],0,0,0)。RC[1]＝1，RC[j]＝2＊RC[j－1]，且乘法定义在域 GF(2^8)上。以 16 进制表示的 RC[j]的值为

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

示例 5-1 假设 N_k＝4，输入的 128 位密钥如下：

密钥＝2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

试计算密钥扩展。

解 输入的密钥为 128 位，因此 N_b＝4，N_r＝10。

第 1 轮的密钥为输入密钥，有

w0＝2b7e1516， w1＝28aed2a6， w3＝abf71588， w4＝09cf4f3c

其余轮密钥计算结果如下：

i (dec)	temp	RotWord 运算之后	SubWord 运算之后	Rcon[i/Nk]	与 Rcon 异或之后	w[i-Nk]	w[i]＝temp XOR w[i-Nk]
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1

续表

i (dec)	temp	RotWord 运算之后	SubWord 运算之后	Rcon[i/Nk]	与 Rcon 异或之后	w[i-Nk]	w[i]=temp XOR w[i-Nk]
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cd42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d					7a96b943	4716fe3e
14	4716fe3e					5935807a	1e237e44
15	1e237e44					7359f67f	6d7a883b
16	6d7a883b	7a883b6d	dac4e23c	08000000	d2c4e23c	3d80477d	ef44a541
17	ef44a541					4716fe3e	a8525b7f
18	a8525b7f					1e237e44	b671253b
19	b671253b					6d7a883b	db0bad00
20	db0bad00	0bad00db	2b9563b9	10000000	3b9563b9	ef44a541	d4d1c6f8
21	d4d1c6f8					a8525b7f	7c839d87
22	7c839d87					b671253b	caf2b8bc
23	caf2b8bc					db0bad00	11f915bc
24	11f915bc	f915bc11	99596582	20000000	b9596582	d4d1c6f8	6d88a37a
25	6d88a37a					7c839d87	110b3efd
26	110b3efd					caf2b8bc	dbf98641
27	dbf98641					11f915bc	ca0093fd
28	ca0093fd	0093fdca	63dc5474	40000000	23dc5474	6d88a37a	4e54f70e
29	4e54f70e					110b3efd	5f5fc9f3
30	5f5fc9f3					dbf98641	84a64fb2
31	84a64fb2					ca0093fd	4ea6dc4f
32	4ea6dc4f	a6dc4f4e	2486842f	80000000	a486842f	4e54f70e	ead27321
33	ead27321					5f5fc9f3	b58dbad2
34	b58dbad2					84a64fb2	312bf560
35	312bf560					4ea6dc4f	7f8d292f

续表

i (dec)	temp	RotWord 运算之后	SubWord 运算之后	Rcon[i/Nk]	与 Rcon 异或之后	w[i-Nk]	w[i]=temp XOR w[i-Nk]
36	7f8d292f	8d292f7f	5da515d2	1b000000	46a515d2	ead27321	ac7766f3
37	ac7766f3					b58dbad2	19fadc21
38	19fadc21					312bf560	28d12941
39	28d12941					7f8d292f	575c006e
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

192 位密钥的轮密钥生成方法、256 位密钥的轮密钥生成方法与 128 位的轮密钥生成方法相同。

5.4 AES 算法实现

AES 算法的实现主要包括四部分：数据初始化、密钥生成、加密和解密。根据 AES 加密和解密的特点，在算法的实现过程中采用了 byte(unsigned char)型数据作为主要的数据类型，并由 4 个字节组成一个字(word)，作为结构体进行处理。

在示例中 AES 算法是针对数据分组长度为 4，密钥分组长度为 4，加密轮数为 10 来设计的，为部分简化了的 AES 算法，这样更有助于理解 AES 算法实现的具体过程。算法实现的基本结构如图 5-8 所示。

AES 类中的各主要变量的作用如下：

cipherKey[16]——原始加密密钥，16 字节的数组，每个字节 8 位，共 128 位。

plaintext[4]——输入的明文，4 字组成的数组，每字 4 个字节，共 128 位。

cipherText[4]——对输入的明文进行加密后得到的密文，4 字组成的数组，每字 4 个字节，共 128 位。

deCipherText[4]——对密文进行解密后的数据，4 字组成的数组，每字 4 个字节，共 128 位。

Nb、Nk 和 Nr——Nb 表示数据分组长度/32，Nk 表示密钥分组长度/32，Nr 表示轮数。具体取值参考表 5-1。在本示例程序中，Nb 为 4，Nk 为 4，Nr 为 10。

Rcon[11]——用于密钥扩展的常量。

wordKey[44]——以字形式表示的扩展后的密钥，供各轮加密过程使用。

Sbox[16][16]——S 盒数据，数据类型为字节型，为 16×16 的二维数组。

invSBox[16][16]——S 盒的逆，数据类型为字节型，为 16×16 的二维数组。用于解密过程。

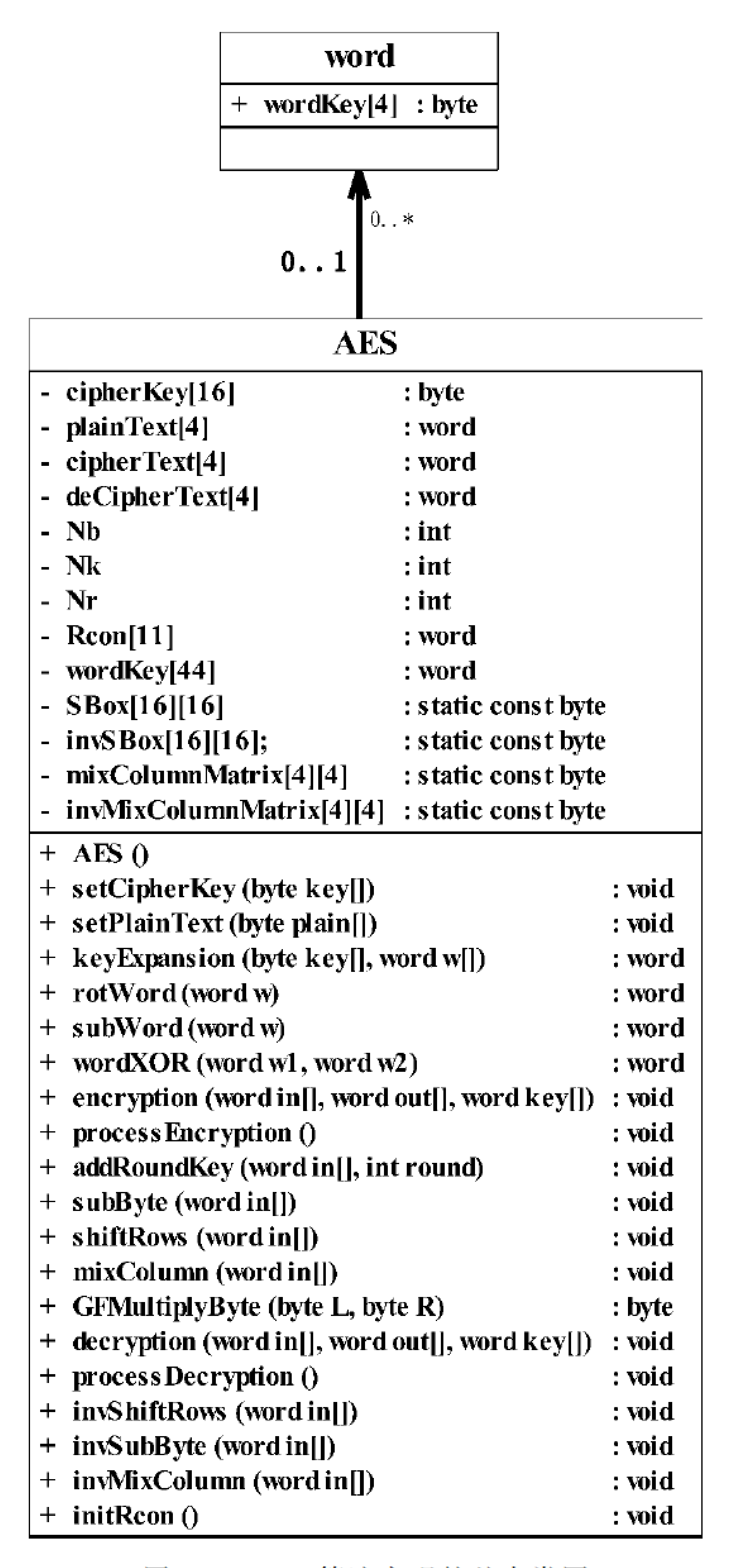


图 5-8 AES 算法实现的基本类图

mixColumnMatrix[4][4]——列混淆常量矩阵，数据类型为字节型，为 4×4 的二维数组。

invMixColumnMatrix[4][4]——列混淆逆常量矩阵，数据类型为字节型，为 4×4 的二维数组。用于解密过程。

详细的类声明见程序清单 5-2。

程序清单 5-2

```
typedef unsigned char byte;
struct word
```

```
{
    byte wordKey[4];
};
class AES
{
    public:
        AES();
        void setCipherKey(byte key[]);
        void setPlainText(byte plain[]);
        void keyExpansion(byte key[],word w[]);
        word rotWord(word w);
        word subWord(word w);
        word wordXOR(word w1,word w2);
        //用于处理加密和解密的各函数
        void encryption(word in[],word out[],word key[]);
        void processEncryption();
        void addRoundKey(word in[],int round);
        void subByte(word in[]);
        void shiftRows(word in[]);
        void mixColumn(word in[]);
        byte GFMultiplyByte(byte L,byte R);
        //用于处理解密的各函数
        void decryption(word in[],word out[],word key[]);
        void processDecryption();
        void invShiftRows(word in[]);
        void invSubByte(word in[]);
        void invMixColumn(word in[]);
        void initRcon();
    private:
        byte cipherKey[16];
        word plainText[4];
        word cipherText[4];
        word deCipherText[4];
        int Nb,Nk,Nr;
        word Rcon[11];
        word wordKey[44];
        static const byte SBox[16][16];
        static const byte invSBox[16][16];
        static const byte mixColumnMatrix[4][4];
        static const byte invMixColumnMatrix[4][4];
};
```

5.4.1 数据初始化

数据初始化包括：明文初始化、初始密钥初始化和加密解密过程中所需的各种常量的

初始化。

(1) 明文初始化——通过函数 void setPlainText(byte plain[])来设置明文，函数参数为字节型数组，函数的输入为 16 字节的明文。函数的定义见程序清单 5-3。

程序清单 5-3

```
01  void AES::setPlainText(byte plain[])
02  {
03      int i;
04      for(i=0;i<16;i++)
05      {
06          plainText[i%4].wordKey[i/4]=plain[i];
07      }
08  }
```

在程序清单 5-3 中的第 6 行代码是将二维数组处理成适合 AES 算法的格式，即状态矩阵(state)格式，若转换成先行后列格式的代码为

```
plainText[i/4].wordKey[i%4]=plain[i];
```

由于 plainText 的数据类型是 word 型数据，word 结构体包含 byte 型数组，该数组的大小为 4，类型为 byte 型。例如，当输入的明文是 0～4 字节时，那么第 1 个字节存储到 plainText[0][0]，第 2 个字节存储到 plainText[0][1]，以此类推，将读入的明文数据转换为便于 AES 算法易于处理的格式。通过程序清单 5-3 的转换，输入的明文转换为矩阵格式。

(2) 初始密钥初始化——初始密钥通过函数 void setCipherKey(byte key[])来初始化，函数的参数为字节型数组，函数的输入为 16 字节的密钥。函数的定义见程序清单 5-4。

程序清单 5-4

```
01  void AES::setCipherKey(byte key[])
02  {
03      int i;
04      for(i=0;i<16;i++)
05      {
06          cipherKey[i]=key[i];
07      }
08      keyExpansion(cipherKey,wordKey);
09  }
```

程序清单 5-4 将密钥参数获得后，首先将输入的密钥赋给对应类中的密钥，这部分工作由代码行第 4 行到第 7 行完成，然后通过密钥扩展函数 keyExpansion()进行密钥扩展，密钥扩展将在 5.4.2 节中介绍。

(3) Rcon 初始化——Rcon 数组的初始化通过 void initRcon()函数来完成，函数的定义见程序清单 5-5。

程序清单 5-5

```
01  void AES::initRcon()
```

```
{
    int i,j;
    for(i=0;i<11;i++)
    {
        for(j=0;j<4;j++)
        {
            Rcon[i].wordKey[j]=0x0;
        }
    }
    Rcon[1].wordKey[0]=0x01;
    Rcon[2].wordKey[0]=0x02;
    Rcon[3].wordKey[0]=0x04;
    Rcon[4].wordKey[0]=0x08;
    Rcon[5].wordKey[0]=0x10;
    Rcon[6].wordKey[0]=0x20;
    Rcon[7].wordKey[0]=0x40;
    Rcon[8].wordKey[0]=0x80;
    Rcon[9].wordKey[0]=0x1b;
    Rcon[10].wordKey[0]=0x36;
}
```

由于 Rcon 是固定值，程序中直接给出相应的值，该值也可以通过 GF(2^8)域的乘法函数 byte GFMultiplyByte(byte L，byte R)计算获得。

(4) 列混淆和列混淆求逆常量的初始化——见程序清单 5-6。

程序清单 5-6

```
const byte AES::mixColumnMatrix[4][4]={
    {0x02,0x03,0x01,0x01},
    {0x01,0x02,0x03,0x01},
    {0x01,0x01,0x02,0x03},
    {0x03,0x01,0x01,0x02}};
const byte AES::invMixColumnMatrix[4][4]={
    {0x0e,0x0b,0x0d,0x09},
    {0x09,0x0e,0x0b,0x0d},
    {0x0d,0x09,0x0e,0x0b},
    {0x0b,0x0d,0x09,0x0e}};
```

(5) S 盒和 S 盒逆的初始化——见程序清单 5-7。

程序清单 5-7

```
//初始化 S 盒
const byte AES::SBox[16][16]={
{0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76},
{0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0},
{0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15},
{0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75},
```

```
{0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84},
{0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF},
{0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8},
{0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2},
{0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73},
{0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB},
{0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79},
{0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08},
{0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A},
{0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E},
{0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF},
{0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16}};
//初始化 S 盒逆
const byte AES::invSBox[16][16]={
{0x52,0x09,0x6A,0xD5,0x30,0x36,0xA5,0x38,0xBF,0x40,0xA3,0x9E,0x81,0xF3,0xD7,0xFB},
{0x7C,0xE3,0x39,0x82,0x9B,0x2F,0xFF,0x87,0x34,0x8E,0x43,0x44,0xC4,0xDE,0xE9,0xCB},
{0x54,0x7B,0x94,0x32,0xA6,0xC2,0x23,0x3D,0xEE,0x4C,0x95,0x0B,0x42,0xFA,0xC3,0x4E},
{0x08,0x2E,0xA1,0x66,0x28,0xD9,0x23,0xB2,0x76,0x5B,0xA2,0x49,0x6D,0x8B,0xD1,0x25},
{0x72,0xF8,0xF6,0x64,0x86,0x68,0x98,0x16,0xD4,0xA4,0x5C,0xCC,0x5D,0x65,0xB6,0x92},
{0x6C,0x70,0x48,0x50,0xFD,0xED,0xB9,0xDA,0x5E,0x15,0x46,0x57,0xA7,0x8D,0x9D,0x84},
{0x90,0xD8,0xAB,0x00,0x8C,0xBC,0xD3,0x0A,0xF7,0xE4,0x58,0x05,0xB8,0xB3,0x45,0x06},
{0xD0,0x2C,0x1E,0x8F,0xCA,0x3F,0x0F,0x02,0xC1,0xAF,0xBD,0x03,0x01,0x13,0x8A,0x6B},
{0x3A,0x91,0x11,0x41,0x4F,0x67,0xDC,0xEA,0x97,0xF2,0xCF,0xCE,0xF0,0xB4,0xE6,0x73},
{0x96,0xAC,0x74,0x22,0xE7,0xAD,0x35,0x85,0xE2,0xF9,0x37,0xE8,0x1C,0x75,0xDF,0x6E},
{0x47,0xF1,0x1A,0x71,0x1D,0x29,0xC5,0x89,0x6F,0xB7,0x62,0x0E,0xAA,0x18,0xBE,0x1B},
{0xFC,0x56,0x3E,0x4B,0xC6,0xD2,0x79,0x20,0x9A,0xDB,0xC0,0xFE,0x78,0xCD,0x5A,0xF4},
{0x1F,0xDD,0xA8,0x33,0x88,0x07,0xC7,0x31,0xB1,0x12,0x10,0x59,0x27,0x80,0xEC,0x5F},
{0x60,0x51,0x7F,0xA9,0x19,0xB5,0x4A,0x0D,0x2D,0xE5,0x7A,0x9F,0x93,0xC9,0x9C,0xEF},
{0xA0,0xE0,0x3B,0x4D,0xAE,0x2A,0xF5,0xB0,0xC8,0xEB,0xBB,0x3C,0x83,0x53,0x99,0x61},
{0x17,0x2B,0x04,0x7E,0xBA,0x77,0xD6,0x26,0xE1,0x69,0x14,0x63,0x55,0x21,0x0C,0x7D}};
```

5.4.2 轮密钥计算

轮密钥的计算通过密钥扩展函数来实现，实现过程的伪码已在程序清单 5-1 中给出，具体的实现过程见程序清单 5-8。

程序清单 5-8

```
void AES::keyExpansion(byte key[],word w[])
{
    int i=0;
    int j,k;
    word temp;
    while(i<Nk)
    {
        for(j=0;j<4;j++)
```

```
09              {
10                  w[i].wordKey[j]=key[j+4*i];
11              }
12              i++;
13          }
14          i=Nk;
15          while(i<Nb*(Nr+1))
16          {
17              temp=w[i-1];
18              if((i%Nk)==0)
19              {
20                  temp=rotWord(temp);
21                  temp=subWord(temp);
22                  temp=wordXOR(temp,Rcon[i/Nk]);
23              }
24              else if(Nk>6 && (i%Nk)==4)        //处理其他可能的情况
25              {
26                  temp=subWord(temp);
27              }
28              w[i]=wordXOR(w[i-Nk],temp);
29              i++;
30          }
31          word tempState[44];
32          for(i=0;i<11;i++)
33          {
34              for(j=0;j<4;j++)
35              {
36                  for(k=0;k<4;k++)
37                  {
38                      tempState[j+i*4].wordKey[k]=w[k+i*4].wordKey[j];
39                  }
40              }
41          }
42          for(i=0;i<44;i++)
43          {
44              for(j=0;j<4;j++)
45              {
46                  w[i].wordKey[j]=tempState[i].wordKey[j];
47              }
48          }
49      }
```

在程序清单 5-8 中，第 6 行到第 13 行代码为计算第 0 轮密钥，第 0 轮的密钥实际上就是输入的密钥，对于本示例程序，由于 $N_k=4$，经过双循环之后，正如将输入的 16 字节的数据转换为密钥使用的 4×4 矩阵格式。第 15 行到第 30 行代码为计算第 1 轮到第 10 轮的密

钥，第 32 行到第 41 行是将密钥的处理方式改为 AES 运算过程中的状态矩阵。

在第 1 轮到第 10 轮的密钥计算过程中还用到了 rotWord()、subWord()和 wordXOR()三个函数。rotWord()函数的功能是实现字循环，将字中的 4 个字节循环左移 1 个字节，subWord()函数的功能是字节代换，利用 S 盒将字中的每个字节进行代换，wordXOR()函数的功能是实现字与 Rcon 的“异或”。

rotWord()函数的代码见程序清单 5-9。

程序清单 5-9

```
01  word AES::rotWord(word w)
02  {
03      int i;
04      word temp;
05      for(i=0;i<4;i++)
06      {
07          temp.wordKey[(i+3)%4]=w.wordKey[i];
08      }
09      return temp;
10  }
```

代码行的第 5 行到第 8 行实现循环左移一个字节的功能，输入的 word 型数据包含 4 个字节，假设为 byte0||byte1||byte2||byte3，那么，经过循环左移后得到的数据为 byte3||byte0||byte1||byte2。

subWord()函数的代码见程序清单 5-10。

程序清单 5-10

```
word AES::subWord(word w)
{
    int i;
    byte L,R;
    for(i=0;i<4;i++)
    {
        L=w.wordKey[i]>>4;             //获取输入的高 4 位
        R=w.wordKey[i]&0x0F;           //获取输入的低 4 位
        w.wordKey[i]=SBox[L][R];
    }
    return w;
}
```

subWord()函数是通过获得字节的高 4 位和低 4 位来作为 S 盒数组的行列位置，然后确定代换的值。高 4 位直接通过右移 4 位来获得，低 4 位与 0x0F 进行“&”运算，将高 4 位置为“0”来获得，0x0F 用二进制可表示为 00001111，当与字节型数据进行“&”运算时，获得数据的高 4 位均为“0”，而低 4 位数据保持不变。

wordXOR()函数的代码见程序清单 5-11。

程序清单 5-11

```
word AES::wordXOR(word w1,word w2)
{
    int i;
    word temp;
    for(i=0;i<4;i++)
    {
        temp.wordKey[i]=w1.wordKey[i]^w2.wordKey[i];
    }
    return temp;
}
```

wordXOR()函数通过将两个 word 型数据中的每个字节进行“异或”处理，最终获得两个字的“异或”，并作为结果返回。

5.4.3 AES 加密过程的实现

AES 算法的加密过程通过函数 void encryption(word in[],word out[],word key[])来实现，函数的具体代码见程序清单 5-12。

程序清单 5-12

```
void AES::encryption(word in[],word out[],word key[])
{
    int i,j,k;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            out[i].wordKey[j]=in[i].wordKey[j];
        }
    }
    addRoundKey(out,0);
    for(i=1;i<10;i++)
    {
        subByte(out);
        shiftRows(out);
        mixColumn(out);
        addRoundKey(out,i);
    }
    subByte(out);
    shiftRows(out);
    addRoundKey(out,10);
}
```

在加密过程中，首先进行一轮轮密钥加运算，然后进行 9 轮完全相同的加密过程，包含

字节代换、行移位、列混淆和轮密钥加，最后进行字节代换、行移位和轮密钥加运算完成整个加密过程，第 4 行到第 10 行代码是将输入赋给加密输出，然后通过第 11 行代码进行轮密钥加运算，第 12 行到第 18 行代码进行 9 轮完全相同的加密运算，再通过第 19 行的字节代换、第 20 行的行移位和第 21 行的轮密钥加运算完成整个加密过程。

在加密过程中用到了 subByte()函数来实现字节代换功能、shiftRows()来实现行移位功能、mixColumn()函数来实现列混淆功能、addRoundKey()函数来实现轮密钥加功能。

字节代换 subByte()函数的具体实现代码见程序清单 5-13。

程序清单 5-13

```
01  void AES::subByte(word in[])
02  {
03      int i,j;
04      byte L,R;
05      for(i=0;i<4;i++)
06      {
07          for(j=0;j<4;j++)
08          {
09              L=in[i].wordKey[j]>>4;
10              R=in[i].wordKey[j]&0x0F;
11              in[i].wordKey[j]=SBox[L][R];
12          }
13      }
14  }
```

在加密过程中使用的字节代换与密钥计算过程中使用的字节代换函数相似，在加密过程中使用的字节代换的函数参数是“字”数组，而在密钥计算过程中的函数参数是“字”。两个函数的高 4 位和低 4 位的取值方法完全相同。第 9 行和第 10 行代码分别获得 S 盒的行和列所在的位置，即输入数据对应字节的高 4 位和低 4 位，第 11 行代码获取 S 盒相应位置的数据，并对应赋给输入“字”的相应字节。

行移位 shiftRows()函数的具体实现代码见程序清单 5-14。

程序清单 5-14

```
void AES::shiftRows(word in[])
{
    int i,j;
    word temp[4];
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            temp[i].wordKey[(j+(4-i))%4]=in[i].wordKey[j];
        }
    }
    for(i=0;i<4;i++)
```

```
13         {
14             for(j=0;j<4;j++)
15             {
16                 in[i].wordKey[j]=temp[i].wordKey[j];
17             }
18         }
19     }
```

行移位 shiftRows()函数的实现方法与密钥生成中字循环移位方法类似，但在加密过程中是对输入的矩阵进行行移位，每行移位的多少还不相同。第 5 行到第 11 行代码实现了 AES 算法中的移位要求，即：第 1 行不移位，第 2 行循环左移 1 位，第 2 行循环左移 2 位，第 3 行循环左移 3 位。例如，当 i=0 时，(j+(4−i))%4 的作用与 j%4 的作用相同，因此整个过程不移位，当 i=1 时，(j+(4−i))%4 的作用相当于(j+3)%4，正好完成循环左移 1 位，其他步骤以此类推。

列混淆 mixColumn()函数的实现代码见程序清单 5-15。

程序清单 5-15

```
01 void AES::mixColumn(word in[])
02 {
03     word result[4];
04     int i,j,k;
05     for(i=0;i<4;i++)
06     {
07         for(j=0;j<4;j++)
08         {
09             result[i].wordKey[j]=
10                 GFMultiplyByte(mixColumnMatrix[i][0],in[0].wordKey[j]);
11             for(k=1;k<4;k++)
12             {
13               result[i].wordKey[j]^=
14                   GFMultiplyByte(mixColumnMatrix[i][k],in[k].wordKey[j]);
15             }
16         }
17     }
18     for(i=0;i<4;i++)
19     {
20         for(j=0;j<4;j++)
21         {
22             in[i].wordKey[j]=result[i].wordKey[j];
23         }
24     }
25 }
```

代码行第 5 行到第 17 行是完成公式(5-5)的运算，以完成第 0 行第 j 列的数据来说明，相应的计算方法如下：

$$b_{0,j} = (2 \cdot a_{0,j}) \oplus (3 \cdot a_{1,j}) \oplus a_{2,j} \oplus a_{3,j}$$

式中的计算均为在 GF(2^8)有限域中的运算，在计算过程中首先进行第 1 项，这步运算通过代码行第 9 行、第 10 行代码完成，在计算得到第 2 项数据之后，再计算其余项，并与前 1 项的结果进行“异或”运算，直到完成全部运算。在列混淆计算过程中使用的乘法运算和加法运算都是在 GF(2^8)有限域中进行的运算。

列混淆函数使用 GFMultiplyByte()函数实现在 GF(2^8)有限域上的乘法运算，函数的具体实现代码见程序清单 5-16。

程序清单 5-16

```
01 byte AES::GFMultiplyByte(byte L,byte R)
02 {
03     byte temp[8];
04     byte result=0x00;
05     temp[0]=L;
06     int i;
07     for(i=1;i<8;i++)
08     {
09         if(temp[i-1]>=0x80)
10         {
11             temp[i]=(temp[i-1]<<1)^0x1b;      //与 00011011 异或
12         }
13         else
14         {
15             temp[i]=temp[i-1]<<1;
16         }
17     }
18     for(i=0;i<8;i++)
19     {
20         if(int((R>>i)&0x01)==1)
21         {
22             result^=temp[i];
23         }
24     }
25     return result;
26 }
```

GFMultiplyByte()函数实现了在 GF(2^8)有限域内的乘法运算，乘法运算通过第 9 行到第 24 行代码实现，实现的方法是利用了 GF(2^8)有限域乘法的特点来进行的。在 AES 算法中，数据均以 byte 格式进行处理，用二进制的形式可以表示为：$\{b_7,b_6,b_5,b_4,b_3,b_2,b_1,b_0\}$，使用多项式的形式可以表示为

$$f(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0 = \sum_{i=0}^{7} b_ix^i$$

例如，(00100101)可以表示为 x^5+x^2+1。而计算 $x \times f(x)$则具有以下特征：

$$x \times f(x) = \begin{cases} (b_6 b_5 b_4 b_3 b_2 b_1 b_0 0), & b_7 = 0 \\ (b_6 b_5 b_4 b_3 b_2 b_1 b_0 0) \oplus (00011011), & b_7 = 1 \end{cases}$$

上式表示当 $b_7=0$ 时，只需左移一位就可以得到计算结果，当 $b_7=1$ 时（输入字节大于等于 0x80），则先左移一位，然后与（00011011）进行“异或”运算就可以得到计算结果。（00011011）用十六进制表示为 0x1B。

当 $f(x)$ 乘以高于一次的多项式时，则可以重复使用上述计算过程来完成。

轮密钥加 addRoundKey()函数详细代码见程序清单 5-17。

程序清单 5-17

```
void AES::addRoundKey(word in[],int round)
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            in[i].wordKey[j]^=wordKey[i+4*round].wordKey[j];
        }
    }
}
```

轮密钥加函数所使用的子密钥由具体的轮数确定，在函数中通过参数 round 来传递具体使用哪一轮的子密钥。

5.4.4 AES 解密过程的实现

AES 算法的解密过程通过函数 void decryption(word in[],word out[],word key[]) 来实现，函数的详细代码见程序清单 5-18。

程序清单 5-18

```
void AES::decryption(word in[],word out[],word key[])
{
    int i,j,k;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            out[i].wordKey[j]=in[i].wordKey[j];
        }
    }
    addRoundKey(out,10);
    for(i=9;i>0;i--)
    {
        invShiftRows(out);
        invSubByte(out);
        addRoundKey(out,i);
```

```
            invMixColumn(out);
        }
        invShiftRows(out);
        invSubByte(out);
        addRoundKey(out,0);
}
```

解密过程的子密钥使用顺序与加密过程使用子密钥的顺序相反，在解密过程中首先使用最后一轮的子密钥进行轮密钥加运算，然后进行9轮完全相同的解密过程，该过程由4步组成，分别是：行移位求逆、字节代换求逆、轮密钥加和列混淆求逆。行移位求逆通过函数invShiftRows()来实现，字节代换求逆通过函数invSubByte()来实现，轮密钥加通过函数addRoundKey()来实现，该函数就是加密过程中的轮密钥加函数，列混淆求逆通过函数invMixColumn()来实现。在完成9轮相同的解密过程之后，再分别进行行移位求逆、字节代换求逆和轮密钥加运算，完成整个解密过程。

行移位求逆函数invShiftRows()的实现代码见程序清单5-19。

程序清单 5-19

```
void AES::invShiftRows(word in[])
{
    int i,j;
    word temp[4];
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            temp[i].wordKey[(j+i)%4]=in[i].wordKey[j];
        }
    }
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            in[i].wordKey[j]=temp[i].wordKey[j];
        }
    }
}
```

列混淆求逆的实现过程与列混淆的实现过程相似，但移位的方式正好相反，列混淆求逆是循环右移，列混淆是循环左移。

字节代换求逆函数invSubByte()的实现代码见程序清单5-20。

程序清单 5-20

```
void AES::invSubByte(word in[])
{
    int i,j;
```

```
        byte L,R;
        for(i=0;i<4;i++)
        {
            for(j=0;j<4;j++)
            {
                L=in[i].wordKey[j]>>4;          //获取输入的高 4 位
                R=in[i].wordKey[j]&0x0F;        //获取输入的低 4 位
                in[i].wordKey[j]=invSBox[L][R];
            }
        }
}
```

字节代换求逆函数的实现方法与字节代换的实现方法相似，不同点是字节代换中使用了 S 盒，字节代换求逆使用的是 S 盒的逆。

列混淆求逆函数 invMixColumn()的详细代码见程序清单 5-21。

程序清单 5-21

```
void AES::invMixColumn(word in[])
{
        word result[4];
        int i,j,k;
        for(i=0;i<4;i++)
        {
            for(j=0;j<4;j++)
            {
                result[i].wordKey[j]=
                GFMultiplyByte(invMixColumnMatrix[i][0],in[0].wordKey[j]);
                for(k=1;k<4;k++)
                {
                    result[i].wordKey[j]^=
                    GFMultiplyByte(invMixColumnMatrix[i][k],in[k].wordKey[j]);
                }
            }
        }
        for(i=0;i<4;i++)
        {
            for(j=0;j<4;j++)
            {
                in[i].wordKey[j]=result[i].wordKey[j];
            }
        }
}
```

列混淆求逆的实现方法与列混淆的实现方法相似，不同点是列混淆使用了列混淆矩阵，列混淆求逆使用了列混淆求逆矩阵，其他部分完全相同。

5.5 习题与实践题

5.5.1 习题

1. 简要说明 AES 加密算法有哪些特点。
2. 简要说明 AES 加密算法中轮密钥加运算过程的基本原理,并给出示例。
3. 简要说明 AES 加密算法中字节代换运算过程的基本原理,并给出示例。
4. 简要说明 AES 加密算法中行移位变换过程的基本原理。
5. 简要说明 AES 加密算法中列混淆变换过程的基本原理。
6. 简要说明 AES 加密算法中密钥生成的基本原理。

5.5.2 实践题

参考 5.1 节的 AES 的实现方法,实现从文件读取待加密的消息,并将加密后的数据存储到文件。同时,程序还可以从文件中还原密文得到明文。所要处理的明文长度可以直接设为 16 字节。

第 6 章

IDEA 算 法

IDEA 是 International Data Encryption Algorithm 的缩写，即国际数据加密标准。IDEA 算法是由瑞士联邦学院的 Xuejia Lai(来学嘉)和 James Massey 研制开发的对称分组密码，是以 DES 算法为基础发展起来的。

6.1 IDEA 算法原理

IDEA 是一个分组长度为 64 位的分组密码算法，密钥长度为 128 位，同一个算法既可以用于加密，又可用于解密。由于 IDEA 是在美国之外提出并发展起来的，因此避开了美国法律上对加密技术进出口的诸多限制，使得 IDEA 算法的实现技术和书籍都可以自由出版和互相交流，推进了 IDEA 的发展和完善。目前许多开源组织都使用 IDEA 作为加密算法，例如，目前广泛应用的 PGP 就使用了 IDEA 算法作为加密算法。

6.1.1 IDEA 算法的基本结构

IDEA 算法的总体结构见图 6-1。

IDEA 算法用到了三个主要运算方法：按位异或、模 2^{16} 的整数加和模 $2^{16}+1$ 的整数乘。

(1) 按位异或——记为⊕。

(2) 模 2^{16} 的整数加——记为⊞。模 2^{16} 的整数加的运算规则如下：

$$a \boxplus b = a + b \bmod 2^{16} \tag{6-1}$$

(3) 模 $2^{16}+1$ 的整数乘——记为⊙。模 $2^{16}+1$ 的整数乘的运算规则如下：

$$a \odot b = a \cdot b \bmod (2^{16}+1) \tag{6-2}$$

注意：全零的分组代表 2^{16}。

6.1.2 IDEA 算法的加密过程

IDEA 算法的数据输入是 64 位，输入的数据被分解为 4 个 16 位子分组：X_1，X_2，X_3 和 X_4，后续加密过程均针对这 4 个分组进行。这 4 个子分组作为 IDEA 加密的第 1 轮的输入，常用的 IDEA 加密算法共有 8 轮。在每一轮加密过程中，输入的子分组相互之间进行异或、相加和相乘，同时与 6 个 16 位的子密钥进行异或、相加和相乘。在不同的两轮之间，第 2 个和第 3 个子分组进行交换。最后的输出变换是 4 个子分组与 4 个子密钥进行异或。

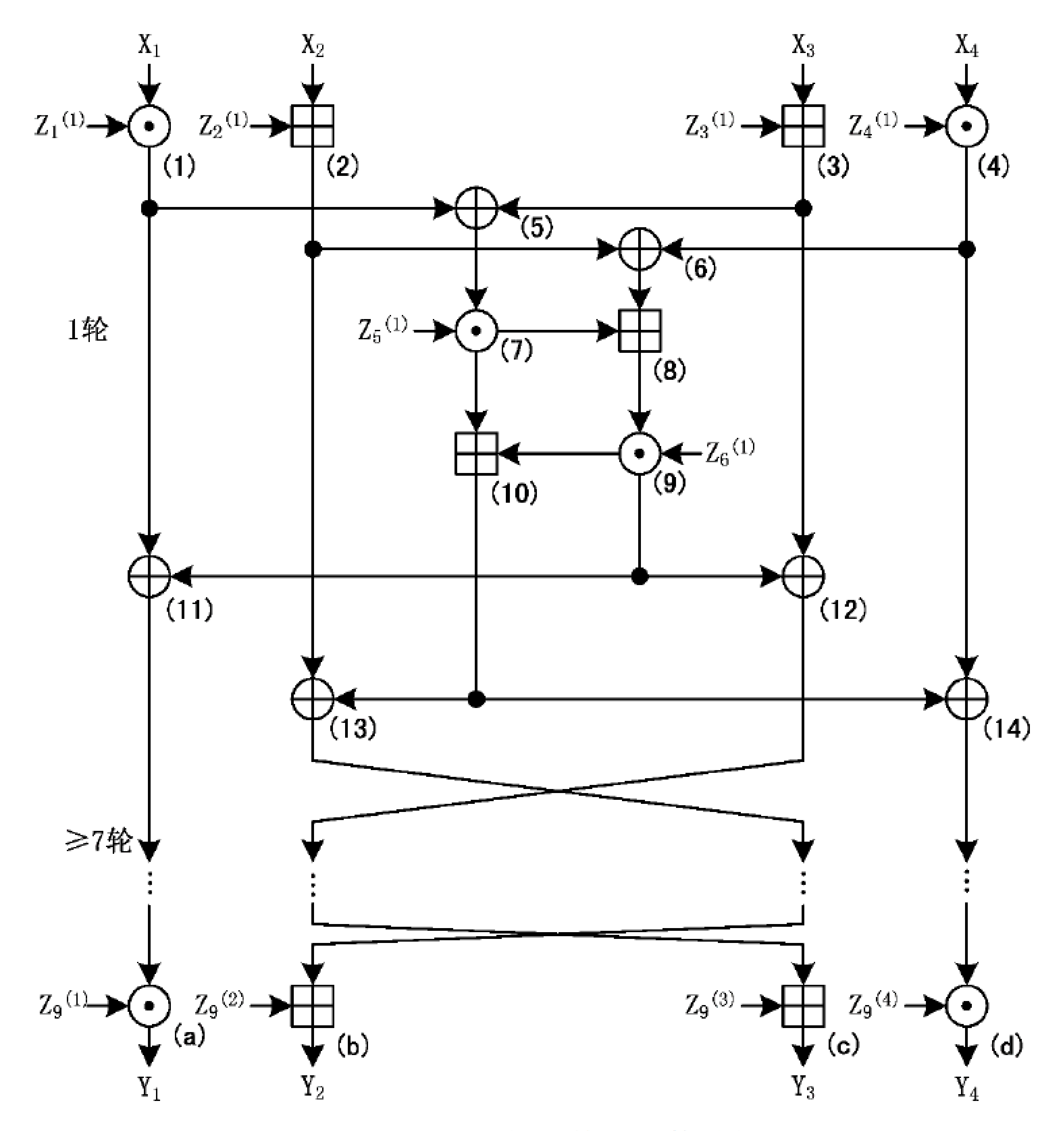


图 6-1 IDEA 加密算法总体结构

对于一轮的加密过程,具体的执行步骤如下:

(1) X_1 和第 1 个子密钥相乘。

(2) X_2 和第 2 个子密钥相加。

(3) X_3 和第 3 个子密钥相加。

(4) X_4 和第 4 个子密钥相乘。

(5) 将第(1)步和第(3)步运算的结果相异或。

(6) 将第(2)步和第(4)步运算的结果相异或。

(7) 将第(5)步的运算结果与第 5 个子密钥相乘。

(8) 将第(6)步的运算结果与第(7)步的运算结果相加。

(9) 将第(8)步的运算结果与第 6 个子密钥相乘。

(10) 将第(7)步和第(9)步的运算结果相加。

(11) 将第(1)步和第(9)步的运算结果相异或。

(12) 将第(3)步和第(9)步的运算结果相异或。

(13) 将第(2)步和第(10)步的运算结果相异或。

(14) 将第(4)步和第(10)步的运算结果相异或。

以上的执行步骤与图 6-1 中标示的步骤相同。在(11)、(12)、(13)和(14)轮计算中得到 4 个子分组,将(12)和(13)得到的两个子分组进行交换后得到下一轮的 4 个子分组。

在完成 8 轮运算之后，还有一步最终变换，即图 6-1 中的(a)、(b)、(c)和(d)运算，这四步最终运算为：

(a) 将最后一轮输出的 X_1 与第 9 轮密钥的第 1 个子密钥相乘。

(b) 将最后一轮输出的 X_2 与第 9 轮密钥的第 2 个子密钥相加。

(c) 将最后一轮输出的 X_3 与第 9 轮密钥的第 3 个子密钥相加。

(d) 将最后一轮输出的 X_4 与第 9 轮密钥的第 4 个子密钥相乘。

6.1.3 子密钥的生成

IDEA 加密算法一共用到 52 个子密钥，8 轮加密(或解密)过程每轮需要 6 个子密钥，共 48 个子密钥，另有 4 个子密钥用于输出。

IDEA 输入的密钥为 128 位，将输入的 128 位密钥分成 8 个 16 位子密钥，这 8 个子密钥是算法的第一批子密钥，分别是第 1 轮的 6 个子密钥和第 2 轮的第 1 个和第 2 个子密钥。然后将密钥循环左移 25 位，再分割成 8 个子密钥，前面 4 个在第 2 轮，后面 4 个在第 3 轮。将密钥再循环左移 25 位生成另外 8 个子密钥，依次循环直到生成所有的子密钥。子密钥的生成过程也可以先将所有的子密钥生成完毕，然后再分解到每一轮。在程序设计中，一般采用先生成 52 个子密钥，然后将 52 个子密钥分配到各轮。

IDEA 加密密钥具体的扩展方法如表 6-1 所示。其中各编号是输入密钥所在“位”位置的顺序编号，每一轮生成密钥的数量为 8 个，并用于不同的加密轮次。

表 6-1 IDEA 加密密钥扩展表

轮	K_1	K_2	K_3	K_4	K_5	K_6
1	0-15	16-31	32-47	48-63	64-79	80-95
2	96-111	112-127	25-40	41-56	57-72	73-88
3	89-104	105-120	121-8	9-24	50-65	66-81
4	82-97	98-113	114-1	2-17	18-33	34-49
5	75-90	91-106	107-122	123-10	11-26	27-42
6	43-58	59-74	100-115	116-3	4-19	20-35
7	36-51	52-67	68-83	84-99	125-12	13-28
8	29-44	45-60	61-76	77-92	93-108	109-124
9	22-37	38-53	54-69	70-85		

IDEA 算法的解密密钥要么是加密密钥的加法逆，要么是乘法逆(例如：模 2^{16} 的乘法逆)。解密密钥的计算量相对要大一点。

加密子密钥和解密子密钥的对照关系如表 6-2 所示。其中 $Z_i^{(j)^{-1}}$ 表示 $Z_i^{(j)}$ 的乘法逆元，由于 $2^{16}+1$ 是一个素数，因此每个非零整数都存在模 $2^{16}+1$ 的乘法逆元，这个关系可以表示为

表 6-2 IDEA 加密子密钥与解密子密钥的对照关系

轮数	加密子密钥	解密子密钥
1	$Z_1^{(1)}$ $Z_2^{(1)}$ $Z_3^{(1)}$ $Z_4^{(1)}$ $Z_5^{(1)}$ $Z_6^{(1)}$	$Z_1^{(9)^{-1}}$ $-Z_2^{(9)}$ $-Z_3^{(9)}$ $Z_4^{(9)^{-1}}$ $Z_5^{(8)}$ $Z_6^{(8)}$
2	$Z_1^{(2)}$ $Z_2^{(2)}$ $Z_3^{(2)}$ $Z_4^{(2)}$ $Z_5^{(2)}$ $Z_6^{(2)}$	$Z_1^{(8)^{-1}}$ $-Z_3^{(8)}$ $-Z_2^{(8)}$ $Z_4^{(8)^{-1}}$ $Z_5^{(7)}$ $Z_6^{(7)}$
3	$Z_1^{(3)}$ $Z_2^{(3)}$ $Z_3^{(3)}$ $Z_4^{(3)}$ $Z_5^{(3)}$ $Z_6^{(3)}$	$Z_1^{(7)^{-1}}$ $-Z_3^{(7)}$ $-Z_2^{(7)}$ $Z_4^{(7)^{-1}}$ $Z_5^{(6)}$ $Z_6^{(6)}$
4	$Z_1^{(4)}$ $Z_2^{(4)}$ $Z_3^{(4)}$ $Z_4^{(4)}$ $Z_5^{(4)}$ $Z_6^{(4)}$	$Z_1^{(6)^{-1}}$ $-Z_3^{(6)}$ $-Z_2^{(6)}$ $Z_4^{(6)^{-1}}$ $Z_5^{(5)}$ $Z_6^{(5)}$
5	$Z_1^{(5)}$ $Z_2^{(5)}$ $Z_3^{(5)}$ $Z_4^{(5)}$ $Z_5^{(5)}$ $Z_6^{(5)}$	$Z_1^{(5)^{-1}}$ $-Z_3^{(5)}$ $-Z_2^{(5)}$ $Z_4^{(5)^{-1}}$ $Z_5^{(4)}$ $Z_6^{(4)}$
6	$Z_1^{(6)}$ $Z_2^{(6)}$ $Z_3^{(6)}$ $Z_4^{(6)}$ $Z_5^{(6)}$ $Z_6^{(6)}$	$Z_1^{(4)^{-1}}$ $-Z_3^{(4)}$ $-Z_2^{(4)}$ $Z_4^{(4)^{-1}}$ $Z_5^{(3)}$ $Z_6^{(3)}$
7	$Z_1^{(7)}$ $Z_2^{(7)}$ $Z_3^{(7)}$ $Z_4^{(7)}$ $Z_5^{(7)}$ $Z_6^{(7)}$	$Z_1^{(3)^{-1}}$ $-Z_3^{(3)}$ $-Z_2^{(3)}$ $Z_4^{(3)^{-1}}$ $Z_5^{(2)}$ $Z_6^{(2)}$
8	$Z_1^{(8)}$ $Z_2^{(8)}$ $Z_3^{(8)}$ $Z_4^{(8)}$ $Z_5^{(8)}$ $Z_6^{(8)}$	$Z_1^{(2)^{-1}}$ $-Z_3^{(2)}$ $-Z_2^{(2)}$ $Z_4^{(2)^{-1}}$ $Z_5^{(1)}$ $Z_6^{(1)}$
输出变换	$Z_1^{(9)}$ $Z_2^{(9)}$ $Z_3^{(9)}$ $Z_4^{(9)}$	$Z_1^{(1)^{-1}}$ $-Z_2^{(1)}$ $-Z_3^{(1)}$ $Z_4^{(1)^{-1}}$

$$Z_i^{(j)} \cdot Z_i^{(j)^{-1}} \equiv 1 \bmod (2^{16}+1) \quad 或 \quad Z_i^{(j)} \odot Z_i^{(j)^{-1}} = 1 \tag{6-3}$$

$-Z_i^{(j)}$ 表示 $Z_i^{(j)}$ 的加法逆元,这个关系可以表示为

$$-Z_i^{(j)} + Z_i^{(j)} \equiv 0 \bmod 2^{16} \quad 或 \quad -Z_i^{(j)} \boxplus Z_i^{(j)} = 0 \tag{6-4}$$

6.2 IDEA 算法实现

IDEA 算法的实现过程主要包括:数据初始化、生成加密密钥、计算解密密钥和加密四个部分。相关数据和函数的声明在 IDEA.h 中,相关实现在 IDEA.cpp 中,具体调用在主函数中。

由于在数据处理过程中用到了 16 位的数据、32 位的数据和字节型的数据,为了方便处理,这部分数据在 IDEA.h 中声明,声明过程如下:

```
typedef unsigned char byte;
typedef unsigned short word16;
typedef unsigned long word32;
```

byte 用于处理字节型数据,word16 用于处理长度为 16 位的数据,word32 用于处理 32 位数据。

软件的具体结构如图 6-2 所示。

IDEA 类图中各主要变量的功能如下:

key[16]——用于获得输入的密钥,数据类型为字节型,处理密钥的长度为 128 位;

plainText[4]——用于存储输入的明文,数据类型为 word16;

cipherText——用于存储加密后得到的密文,数据类型为 word16;

deCipherText[4]——用于存储解密后得到的明文,数据类型为 word16;

encRoundKey[52]——用于存储加密密钥,数据类型为 word16;

IDEA	
- key[16]	: byte
- plainText[4]	: word16
- cipherText	: word16
- deCipherText[4]	: word16
- encRoundKey[52]	: word16
- decRoundKey[52]	: word16
+ setKey (short in[])	: void
+ setPlainText (short in[])	: void
- getEncRoundKey (word16* encRoundKey)	: void
- getDecRoundKey (word16* EK, word16 DK[])	: void
+ encryption (word16 in[], word16 out[], word16* EK)	: void
+ enc ()	: void
+ invMul (word16 x)	: word16
+ mul (word16 x, word16 y)	: word16

图 6-2 IDEA 算法实现的基本类图

decRoundKey[52]——用于存储解密密钥，数据类型为 word16。

IDEA 类的完整声明见程序清单 6-1。

程序清单 6-1

```
class IDEA
{
    public:
        void setKey(byte in[]);
        void setPlainText(byte in[]);
        word16 invMul(word16 x);
        word16 mul(word16 x,word16 y);
        void encryption(word16 in[],word16 out[],word16 * EK);
        void enc();
    private:
        void getEncRoundKey(word16 * encRoundKey);
        void getDecRoundKey(word16 const * EK,word16 DK[]);
        byte key[16];
        word16 plainText[4];
        word16 cipherText[4];
        word16 deCipherText[4];
        word16 encRoundKey[52];
        word16 decRoundKey[52];
        void checkRoundKey();
};
```

6.2.1 数据初始化

在 IDEA 算法的实现过程中，数据初始化包括两部分内容：明文初始化和密钥初始化。明文初始化需要将字节型的明文转换为 word16 型数据，而密钥初始化只需要字节读入字节型的密钥，将字节型密钥转换为 word16 型密钥的过程在轮密钥的计算过程实现。

明文初始化过程由函数 setPlainText()实现，setPlainText()函数的具体细节见程序清单 6-2。

程序清单 6-2

```
01 void IDEA::setPlainText(byte in[])
02 {
03     int i;
04     for(i=0;i<8;i+=2)
05     {
06         plainText[i/2]=(in[i]<<8)+in[i+1];
07     }
08 }
```

由于输入的数据是 8 个 byte 型数据，数据长度为 8 为，而用于加密处理的数据是 16 位的 word16 型数据，因此在获取明文时需将数据进行转换，将两个输入数据合并成一个数据。合并的方法为将第 1 个数据左移 8 位并与第 2 个数据相加，程序清单 6-2 中的第 6 行代码完成了这一功能。

初始化密钥通过函数 setKey()实现，setKey()函数的具体内容见程序清单 6-3。

程序清单 6-3

```
void IDEA::setKey(byte in[])
{
    int i;
    for(i=0;i<16;i++)
    {
        key[i]=in[i];
    }
    getEncRoundKey(encRoundKey);
    getDecRoundKey(encRoundKey,decRoundKey);
}
```

初始化密钥的过程由将密钥数据直接读入、计算加密密钥、计算解密密钥三部分组成，计算加密密钥函数 getEncRoundKey()和计算解密密钥函数 getDecRoundKey()将在 6.2.2 节具体说明。

6.2.2 密钥生成

IDEA 算法的密钥生成包括加密密钥的生成和解密密钥的生成，加密密钥的生成通过函数 getEncRoundKey()实现，具体代码见程序清单 6-4。

程序清单 6-4

```
void IDEA::getEncRoundKey(word16 * encRoundKey)
{
    int i,j;
    for(i=0,j=0;j<8;j++)
```

```
05        {
06            encRoundKey[j]= (key[i]<<8)+key[i+1];
07            i+=2;
08        }
09        for(i=0;j<52;j++)
10        {
11            i++;
12            encRoundKey[i+7]=encRoundKey[i&7]<<9|encRoundKey[(i+1) & 7]>>7;
13            encRoundKey+=i&8;
14            i&=7;
15        }
16    }
```

在程序清单 6-4 中第 4 行到第 8 行代码是将 byte 型密钥转换为 word16 型密钥，输入的 16 个字节型密钥直接转换为 word16 型密钥，即加密密钥的第 0 个密钥到第 8 个密钥，代码行的第 6 行实现将两个字节型数据转换为一个 word16 型数据。

第 9 行到第 15 行代码用于计算其余加密密钥。从第 9 个加密密钥起，每 8 个加密密钥相当于前 8 个密钥循环左移 25 位，图 6-3 为第 9 个(索引值为 8)的密钥的计算方法。

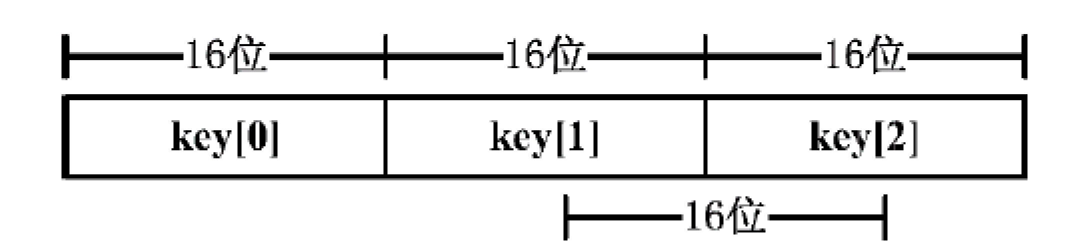


图 6-3　密钥计算方法示意图

第 9 个加密密钥实际上是第 2 个密钥的低 9 位与第 3 个密钥的高 7 位组成，具体实现通过将第 2 个密钥左移 9 位，将第 3 个密钥右移 7 位，再将两者进行“|”运算便得到第 9 个密钥，其他密钥的计算以此类推。例如：假设第 2 个密钥密钥是(1010111100010101)，第 3 个密钥是(1011001000111010)，则第 9 个密钥计算过程为：

(1010111100010101)<<9→(**0010101**000000000)

(1011001000111010)>>7→(0000000**101100100**)

(**0010101**000000000)|(0000000**101100100**)→(**0010101101100100**)

计算解密密钥通过函数 getDecRoundKey()实现，函数具体代码见程序清单 6-5。

程序清单 6-5

```
void IDEA::getDecRoundKey(word16 const * EK,word16 DK[])
{
    int i;
    word16 temp[52];          //计算用临时密钥组
    word16 t1,t2,t3;          //计算用临时变量
    word16 * p=temp+52;       //52 为密钥数量
    t1=invMul(* EK++);
    t2=- * EK++;
    t3=- * EK++;
    * --p=invMul(* EK++);
```

```
    *--p=t3;
    *--p=t2;
    *--p=t1;
    for(i=0;i<7;i++)
    {
        t1=*EK++;
        *--p=*EK++;
        *--p=t1;
        t1=invMul(*EK++);
        t2=-*EK++;
        t3=-*EK++;
        *--p=invMul(*EK++);
        *--p=t2;
        *--p=t3;
        *--p=t1;
    }
    t1=*EK++;
    *--p=*EK++;
    *--p=t1;
    t1=invMul(*EK++);
    t2=-*EK++;
    t3=-*EK++;
    *--p=invMul(*EK++);
    *--p=t3;
    *--p=t2;
    *--p=t1;
    for(i=0,p=temp;i<52;i++)
    {
        *DK++=*p;
        *p++=0;
    }
}
```

表 6-2 中说明了解密密钥与加密密钥的具体关系，以第 1 轮解密密钥为例，第 1 个解密密钥和第 4 个密钥是相应密钥的乘法逆元，第 2 个和第 3 个密钥是相应密钥的加法逆元，第 5 个密钥和第 6 个密钥直接使用了对应的密钥。

计算乘法逆元通过函数 invMul()实现，invMul()函数的具体代码见程序清单 6-6。

程序清单 6-6

```
word16 IDEA::invMul(word16 x)
{
    word16 t0,t1;
    word16 q,y;
    if(x<=1)
    {
```

```
        return x;           //x=0 或 x=1,乘法逆元为其本身
    }
    t1=word16(0x10001L/x);
    y=word16(0x10001L%x);
    if(y==1)
    {
        return (1-t1)&0xFFFF;
    }
    t0=1;
    do{
        q=x/y;
        x=x%y;
        t0+=q* t1;
        if(x==1)
        {
            return t0;
        }
        q=y/x;
        y=y%x;
        t1+=q* t0;
    }while(y!=1);
    return (1-t1)&0xFFFF;
}
```

乘法逆元的计算和 2.3.4 节中的计算方法一样,根据 IDEA 算法的需要,将乘法逆元用 word16 型数据作为函数的返回值返回,并简化了数据处理的方法,将结构体改为单独的变量进行处理。

6.2.3 加密过程和解密过程的实现

加密过程和解密过程通过函数 encryption(word16 in[],word16 out[],word16 * EK)实现,函数参数分别为 word16 的输入、输出和密钥,函数实现的代码见程序清单 6-7。

程序清单 6-7

```
void IDEA::encryption(word16 in[],word16 out[],word16 * EK)
{
    word16 x1,x2,x3,x4,t1,t2;
    x1=in[0];
    x2=in[1];
    x3=in[2];
    x4=in[3];
    int r=8;
    do
    {
        x1=mul(x1,* EK++);
```

```
            x2+=*EK++;
            x3+=*EK++;
            x4=mul(x4,*EK++);
            t2=x1^x3;
            t1=x2^x4;
            t2=mul(t2,*EK++);
            t1=t1+t2;
            t1=mul(t1,*EK++);
            t2=t1+t2;
            x1^=t1;
            x4^=t2;
            t2^=x2;
            x2=x3^t1;
            x3=t2;
        }while(--r);
        x1=mul(x1,*EK++);
        *out++=x1;
        *out++=x3+*EK++;
        *out++=x2+*EK++;
        x4=mul(x4,*EK++);
        *out=x4;
    }
```

加密过程按照 6.1.2 节描述的过程实现，在加密过程中使用了 word16 乘法运算，注意事项为全零的分组代表 2^{16}。乘法运算通过函数 mul()实现，mul()函数的完整代码见程序清单 6-8。

程序清单 6-8

```
word16 IDEA::mul(word16 x,word16 y)
{
    word32 p;
    p=(word32)x*y;
    if(p)
    {
        y=p&0xFFFF;        //取低 16 位
        x=p>>16;
        return (y-x)+(y<x);
    }
    else if(x)
    {
        return 1-y;
    }
    else
    {
```

```
            return 1-x;
        }
}
```

IDEA 算法的加密和解密过程完全一样，只是输入的数据不同，可以在头文件中添加 enc()函数，并在实现文件中具体实现，程序清单 6-9 为加密、解密示例。

程序清单 6-9

```
01 void IDEA::enc()
02 {
03     encryption(plainText,cipherText,encRoundKey);
04     encryption(cipherText,deCipherText,decRoundKey);
05 }
```

代码的第 3 行为加密过程，函数参数分别是输入的明文、输出的密文和加密密钥，代码的第 4 行为解密过程，解密过程使用的函数参数为密文、解密后的明文和解密密钥。

6.2.4 程序测试

若需要测试程序，可以编写相应的测试函数，例如在头文件中加入 IDEATest()函数，该函数用于测试程序结果是否存在问题，IDEATest()函数的具体代码见程序清单 6-10。

程序清单 6-10

```
void IDEA::IDEATest()
{
    ofstream out("ideatest.out");
    out<<"The input key is:"<<endl;
    int i;
    for(i=0;i<16;i++)
    {
        out<<hex<<int(key[i])<<" ";
    }
    out<<endl;
    out<<"The plain text is:"<<endl;
    for(i=0;i<4;i++)
    {
        out<<hex<<plainText[i]<<" ";
    }
    out<<endl;
    out<<"The cipherText is:"<<endl;
    for(i=0;i<4;i++)
    {
        out<<hex<<cipherText[i]<<" ";
    }
    out<<endl;
```

```
23        out<< "The deCipherText is:"<< endl;
24        for(i= 0;i< 4;i+ + )
25        {
26            out<< hex<< deCipherText[i]<< " ";
27        }
28        out<< endl;
29    }
```

代码第 3 行定义保存测试结果的文件，代码第 6 行到第 9 行为输出输入的密钥，代码第 12 行到第 15 行为输出输入的明文，代码第 18 行到第 21 行为输出加密后的密文，代码第 24 行到第 27 行为输出解密后的明文。

具体测试可以通过主函数进行驱动，主函数代码见程序清单 6-11。

程序清单 6-11

```
int main()
{
    IDEA idea;
    byte key[16]= {0x10,0x1A,0x0C,0x0B,0x01,0x11,0x09,0x07,0x32,0xA1,
                   0xB3,0x06,0x23,0x12,0xD3,0xF1};
    idea.setKey(key);
    byte plainText[8]= {0xA7,0x95,0x87,0x23,0x1F,0x2C,0x6D,0x73};
    idea.setPlainText(plainText);
    idea.enc();
    idea.IDEATest();
    return 0;
}
```

在主函数中完成密钥、明文的设置，然后进行加密和解密，再调用 IDEA 类的测试函数完成测试，得到的测试数据如下：

```
The input key is:
10 1a c b 1 11 9 7 32 a1 b3 6 23 12 d3 f1
The plain text is:
a795 8723 1f2c 6d73
The cipherText is:
141c 4811 84d2 24cb
The deCipherText is:
a795 8723 1f2c 6d73
```

经过解密后的数据与输入的明文一致。在这个测试示例中仅对输入的一组数据进行加密和解密，在对文件进行加密和解密时需根据采用的加密和解密模式具体进行。

6.3 习题与实践题

6.3.1 习题

1. 试说明 IDEA 加密算法的密钥生成原理。

2. 试简要说明 IDEA 加密算法的加密过程。

6.3.2 实践题

参考 6.2 节 IDEA 加密算法的实现过程，编程实现 IDEA 加密算法，要求：待加密消息从文件中读取，待加密消息的长度为 64 位。

第 7 章

Blowfish 算法

Blowfish 算法是由 Bruce Schneier 于 1994 年设计的对称分组加密算法，该算法具有快速、紧凑、简单和长度可变等特性，其加密的长度可以从 64 位起，最大可以达到 448 位。Blowfish 算法在设计时就考虑了分组密码存在的一些缺陷，因此，加强了算法的抗差分分析的能力。Blowfish 算法是非专利算法，其算法原理和相关代码都是公开的，因此使其应用得到了较广泛的推广。Blowfish 算法不仅可以单独使用，也可以和其他加密算法结合，进行混合使用，例如：使用 Blowfish 结合 MD5 进行混合加密。

7.1 Blowfish 算法原理

Blowfish 算法是一个 64 位分组密码算法，其密钥长度是可变的。Blowfish 算法的主体结构包括两部分，其一是密钥扩展，其二是加密。Blowfish 算法的基本流程如图 7-1 所示。

图 7-1 中的 P 表示密钥，这个密钥必须在加密和解密之前进行预计算，密钥共有 18 个 32 位的子密钥组成。

典型的 Blowfish 加密算法的输入是 64 位数据，输出也是 64 位数据，计算轮数为 16 轮，加密方法属于典型的 Feistel 结构加密方法。

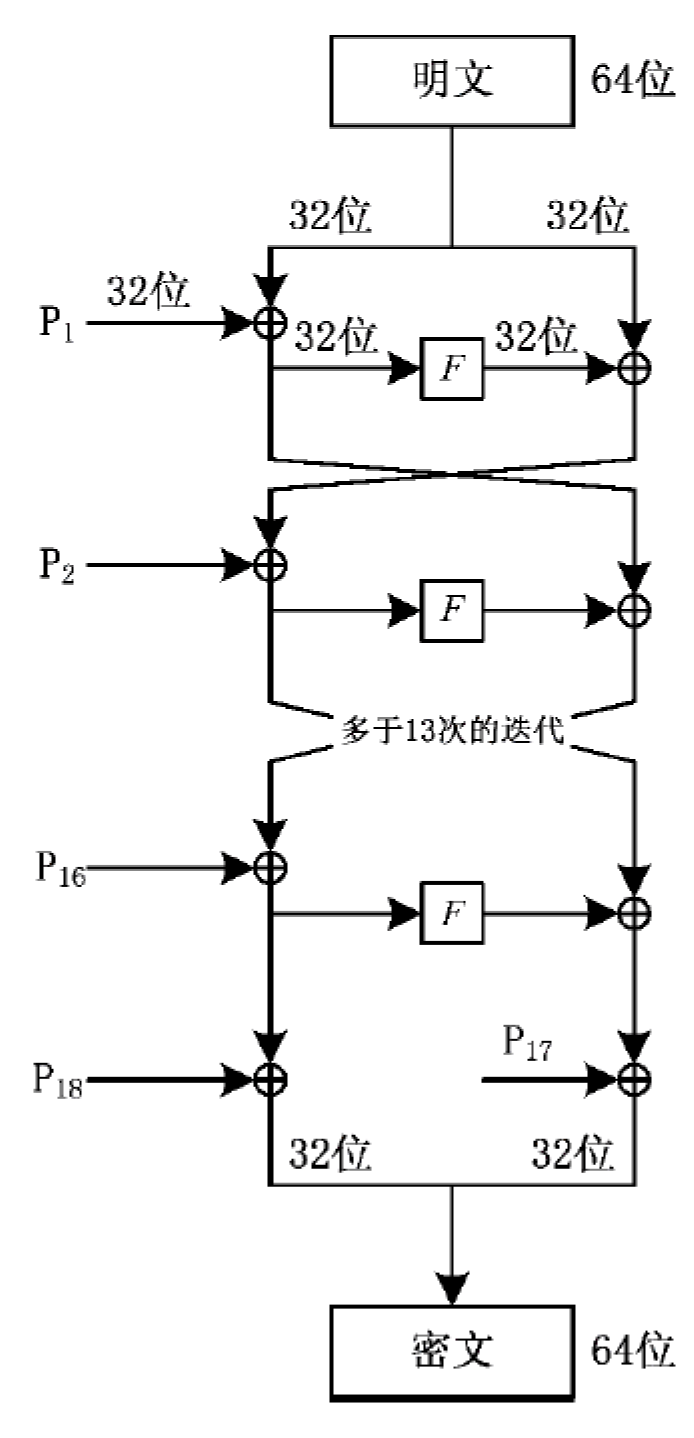


图 7-1　Blowfish 加密算法的基本流程

7.1.1 Blowfish 算法的加解密过程

假设输入的 64 位数据为 X，密钥为 P，则 Blowfish 算法的加密过程可以描述如下：

（1）将输入的 64 位数据分成 32 位的左右两部分，分别记为 X_R 和 X_L。

（2）对于 1 到 16 轮计算：

（a）$X_L = X_L \oplus P_i$

（b）$X_R = F(X_L) \oplus X_R$

（c）如果不是第 16 轮，将 X_R 与 X_L 交换

（3）$X_R = X_R \oplus P_{17}$

(4) $X_L = X_L \oplus P_{18}$

(5) 合并 X_R 和 X_L 得到输出

在加密过程中使用了函数 *F* 来进行加密运算，函数 *F* 的加密过程如图 7-2 所示。

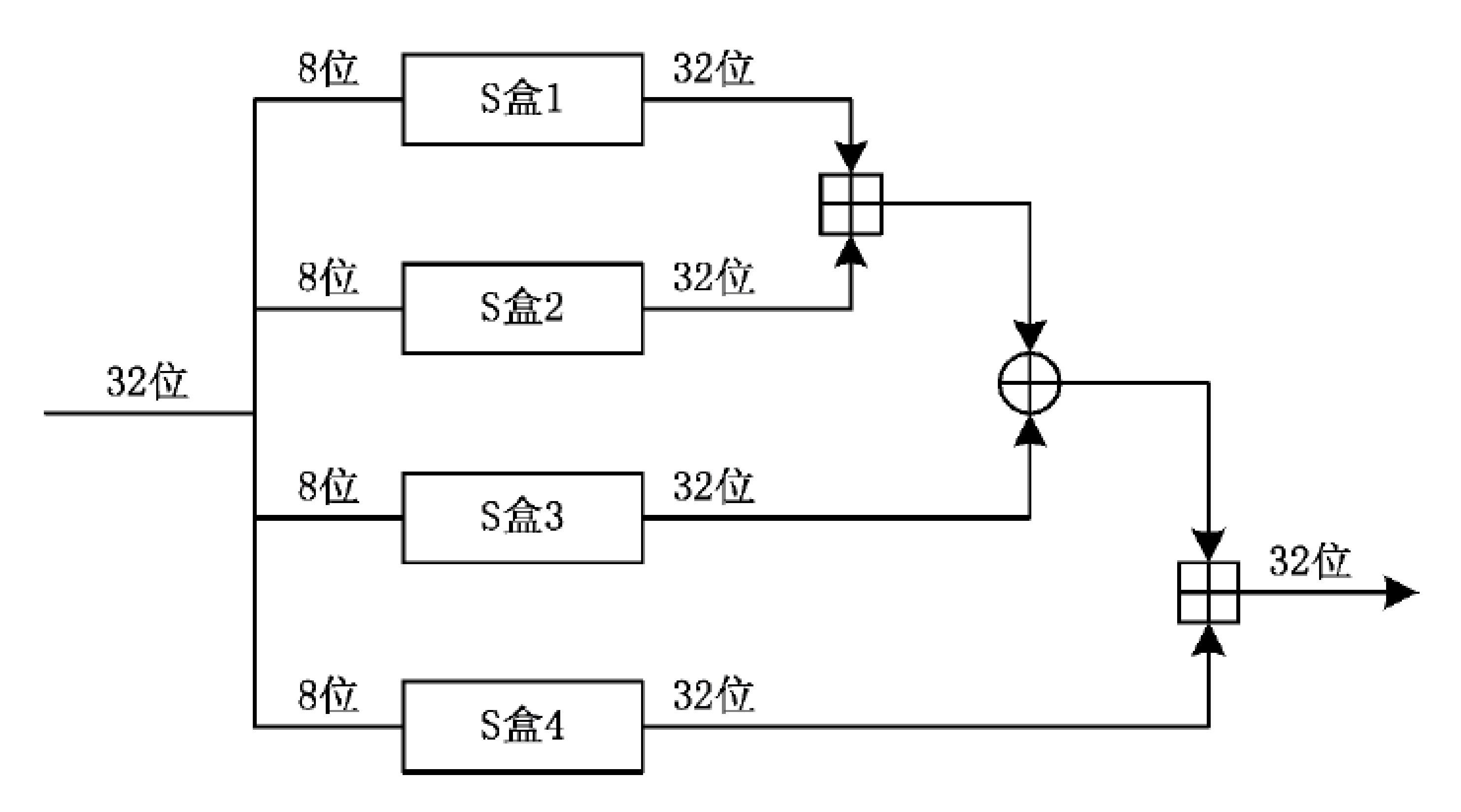


图 7-2 *F* 函数运算示意图

F 函数的运算过程为：

(1) 将 32 位的 X_L 分割成 4 部分 8 位数据，分别为 a,b,c,d。

(2) 计算 $F(X_L)$，$F(X_L)$的计算方法如下：

$$F(X_L) = ((S_{1,a} + S_{2,b} \bmod 2^{32}) \mathrm{XOR} S_{3,c}) + S_{4,d} \bmod 2^{32} \tag{7-1}$$

其中：$S_{1,a}$表示第 1 个 S 盒的第 a 个数据，$S_{2,b}$表示第 2 个 S 盒的第 b 个数据，$S_{3,c}$表示第 3 个 S 盒的第 c 个数据，$S_{4,d}$表示第 4 个 S 盒的第 d 个数据。

在 Blowfish 算法中，共有 4 个 S 盒，每个 S 盒有 256 个数据，每个数据的长度是 32 位。*F* 函数的运算过程中用到的数据是输入数据中的 8 位，该 8 位数据确定取 S 盒中的第几个数据。

Blowfish 的解密过程与加密过程相似，只是以逆序方式使用密钥 $P_1, P_2, \cdots, P_{18}$，其他过程与加密过程一样。

7.1.2 Blowfish 算法的密钥生成

Blowfish 算法的明文输入是 64 位，子密钥 P[i]是 32 位的，初始子密钥根据加密算法的具体细节不同而变换，图 7-1 所示的子密钥数量为 18 个。输入的密钥为字符型密钥，长度可变。Blowfish 使用的子密钥需经过初始计算才能用于数据加密，子密钥的计算流程如下：

(1) 初始化 P 数组，S 盒和全零固定的串。

(2) 用密钥的第一个 32 位与子密钥 P[1]进行"异或"，用密钥的第二个 32 位与子密钥 P[2]进行"异或"，密钥是周期性使用，例如，当输入的密钥为"TwoFish"时，具体使用方法如图 7-3 所示。

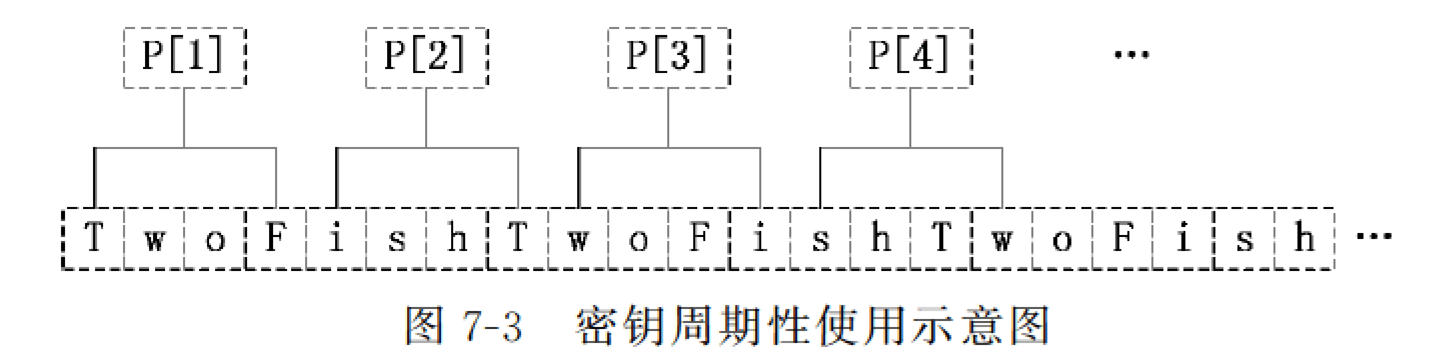


图 7-3 密钥周期性使用示意图

当所有子密钥 P 与密钥异或后结束。

(3) 利用 Blowfish 加密算法对全零串进行加密,加密轮数为子密钥数量,例如有全零串 dataL 和 dataR,其加密流程如图 7-4 所示。

在计算过程中每一轮使用的密钥都不相同,使用的是经过计算以后的 P[i]。

(4) 采用与步骤 3 相同的方法,输入的 dataL 和 dataR 为步骤 3 的最后一步得到的数据,对 S 盒进行同样的计算,在计算 S 盒的过程中用到的 S 盒也是连续变化的。

子密钥和 S 盒的计算过程需要在加密或解密之前完成。

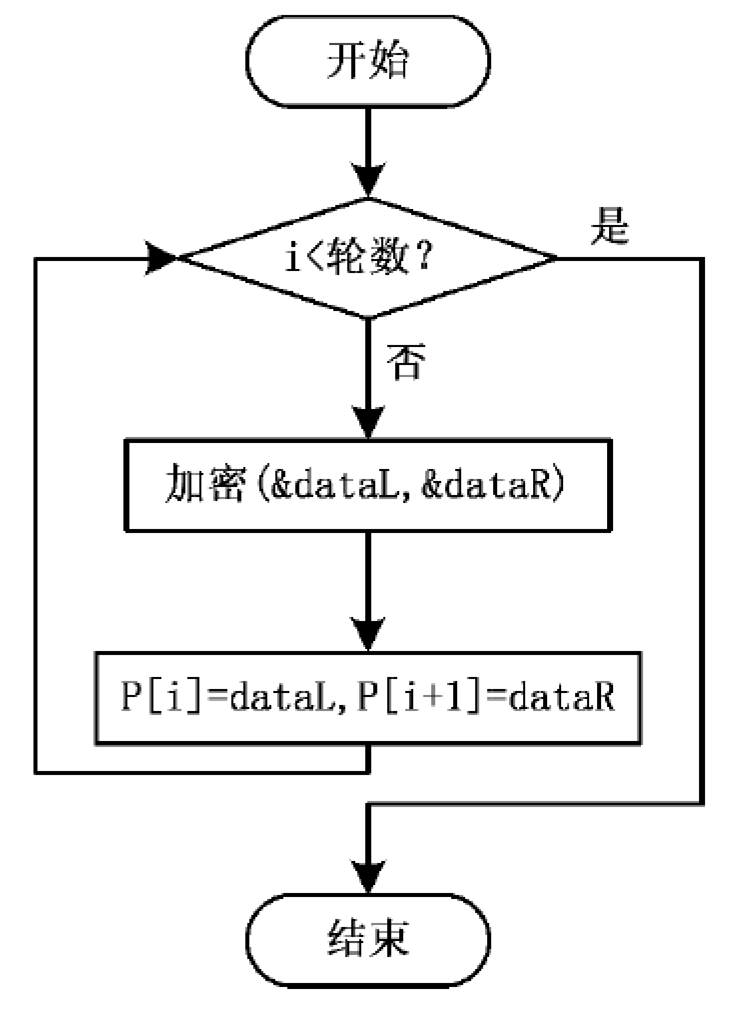


图 7-4　Blowfish 算法子密钥计算流程

7.2　Blowfish 算法实现

Blowfish 算法的特点是子密钥 P 和 S 盒根据密钥的变化而变化,并且在加密或解密之前需用到加密算法来计算子密钥 P 和 S 盒,加密或解密的过程并不使用原始的子密钥 P 和原始 S 盒。

Blowfish 算法实现的基本过程为:

(1) 初始化原始子密钥、原始 S 盒。

(2) 通过输入的可变长度的密钥和加密方法计算子密钥、S 盒。

(3) 加密或解密。

在加密过程中输入的明文是 64 位,但在加密过程中可以分为左右各半,因此只需处理 32 位的数据,在 Blowfish 算法的实现过程中可以用以下方式来处理数据:

```
typedef unsigned char byte;
typedef unsigned long word32;
```

也就是使用 byte 处理 8 位数据,用 word32 处理 32 位数据,这样在数据处理过程中比较简明。

Blowfish 算法实现的类的主要结构如下:

Blowfish	
- originP[18]	: static const word32
- originSBox[4][256]	: static const word32
- P[18]	: word32
- sBox[4][256]	: word32
+ init (unsigned char* key, int keyLength)	: void
+ encryption (word32* xl, word32* xr)	: void
+ decryption (word32* xl, word32* xr)	: void
+ F (word32 x)	: word32
+ Test ()	: void

图 7-5　Blowfish 类的基本结构

Blowfish 类的具体声明见程序清单 7-1。

程序清单 7-1

```
class Blowfish
{
    public:
        Blowfish();
        void init(unsigned char * key,int keyLength);
        void encryption(word32 * xl,word32 * xr);
        void decryption(word32 * xl,word32 * xr);
        word32 F(word32 x);
        void Test();
    private:
        static const word32 originP[18];
        static const word32 originSBox[4][256];
        word32 P[18];
        word32 sBox[4][256];
};
```

在 Blowfish 类中变量主要包括初始子密钥 originP、子密钥 P、初始 S 盒 originSBox 和 S 盒 sBox，函数主要包括初始化函数 init()、加密函数 encryption()、解密函数 decryption() 和 *F* 函数。

7.2.1 加密和解密的实现

在 Blowfish 算法中，生成子密钥和 S 盒均需要用到加密函数。加密函数的具体代码见程序清单 7-2。

程序清单 7-2

```
void Blowfish::encryption(word32 * xl,word32 * xr)
{
    word32 Xl,Xr;
    Xl= * xl;
    Xr= * xr;
    int i;
    word32 temp;
    for(i=0;i<16;i++)
    {
        Xl=Xl^P[i];
        Xr=F(Xl)^Xr;
        if(i<15)
        {
            temp=Xl;
            Xl=Xr;
            Xr=temp;
        }
```

```
18        }
19        Xr=Xr^P[16];
20        Xl=Xl^P[17];
21        *xl=Xl;
22        *xr=Xr;
23    }
```

加密函数将待加密的数据分为左右各半作为参数进行加密，第 8 行代码到第 18 行代码为 16 轮加密运算，在最后一轮运算时，左半部分和右半部分的数据不进行交换。完成 16 轮运算之后再进行余下运算。加密函数既用于加密数据，也用于 P 盒和 S 盒的生成。

解密函数与加密函数的实现方法相同，但使用的 P 盒的顺序与加密函数使用 P 盒的顺序相反。解密函数具体代码见程序清单 7-3。

程序清单 7-3

```
void Blowfish::decryption(word32 *xl,word32 *xr)
{
    word32 Xl,Xr;
    word32 temp;
    int i;
    Xl=*xl;
    Xr=*xr;
    for(i=17;i>1;--i)
    {
        Xl=Xl^P[i];
        Xr=F(Xl)^Xr;
        if(i>2)
        {
            temp=Xl;
            Xl=Xr;
            Xr=temp;
        }
    }
    Xr=Xr^P[1];
    Xl=Xl^P[0];
    *xl=Xl;
    *xr=Xr;
}
```

在加密和解密函数中都用到了 *F* 函数，*F* 函数是 Blowfish 算法的核心之一，*F* 函数具体实现的代码见程序清单 7-4。

程序清单 7-4

```
word32 Blowfish::F(word32 x)
{
    byte a,b,c,d;
```

```
        word32 y;
        d= (byte) (x&0xFF);
        x>>=8;
        c= (byte) (x&0xFF);
        x>>=8;
        b= (byte) (x&0xFF);
        x>>=8;
        a= (byte) (x&0xFF);
        y= sBox[0][a]+ sBox[1][b];
        y= y^sBox[2][c];
        y= y+ sBox[3][d];
        return y;
}
```

F 函数的实现过程：首先将 word32 型 32 位的参数分割成 4 个 8 位的字节型数据，然后，通过这 4 个字节型数据定位在初始 S 盒中的位置，再通过 S 盒的取值计算获得 *F* 函数的运算结果。将 word32 型数据分割成字节型数据的具体方法如图 7-6 所示。

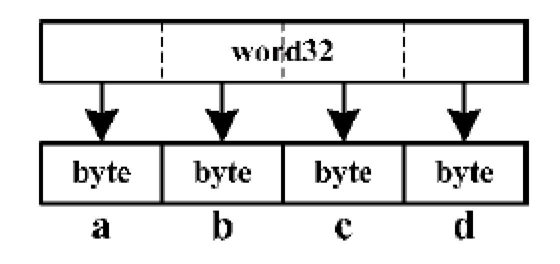


图 7-6 word32→byte 转换示意图

在 *F* 函数实现中充分利用了“&”运算和“>>”(移位)运算，例如：d=(byte)(x&0xFF)；的作用是取 x 的最右 8 位，x>>=8 是将 x 按位右移 8 位，方便进行下一个字节的运算，例如：当 d 运算完毕之后，将 x 右移 8 位，此时再进行(x&0xFF)运算，得到的是字节 c。在数据全部分割完之后再从 S 盒取值并进行相应的运算。在 Blowfish 算法中，S 和为 4×256，在程序中通过 sBox[4][256]，其中，256 正好与 *F* 函数中的 a，b，c，d 对应。

7.2.2 数据初始化

Blowfish 算法数据初始化包括两部分，一部分是初始子密钥 originP 和初始 S 盒 originSBox 的初始化，另一部分是加密或解密数据时与输入密钥相对应的子密钥 P 和 S 盒 sBox 的初始化。初始子密钥 originP 是大小为 18 的一维数组，初始 S 盒 originSBox 为 4×256 的二维数组，在程序示例中作为常量进行处理。

初始子密钥 originP 和初始 S 盒 originSBox 的初始化见程序清单 7-5。

程序清单 7-5

```
const word32 Blowfish::originP[18]= {
        0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344, 0xa4093822, 0x299f31d0,
        0x082efa98, 0xec4e6c89, 0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
        0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917, 0x9216d5d9, 0x8979fb1b};
const word32 Blowfish::originSBox[4][256]= {
        {0xd1310ba6, 0x98dfb5ac, 0x2ffd72db, 0xd01adfb7, 0xb8e1afed, 0x6a267e96,
        0xba7c9045, 0xf12c7f99, 0x24a19947, 0xb3916cf7, 0x0801f2e2, 0x858efc16,
        0x636920d8, 0x71574e69, 0xa458fea3, 0xf4933d7e, 0x0d95748f, 0x728eb658,
        0x718bcd58, 0x82154aee, 0x7b54a41d, 0xc25a59b5, 0x9c30d539, 0x2af26013,
        0xc5d1b023, 0x286085f0, 0xca417918, 0xb8db38ef, 0x8e79dcb0, 0x603a180e,
```

```
0x6c9e0e8b, 0xb01e8a3e, 0xd71577c1, 0xbd314b27, 0x78af2fda, 0x55605c60,
0xe65525f3, 0xaa55ab94, 0x57489862, 0x63e81440, 0x55ca396a, 0x2aab10b6,
0xb4cc5c34, 0x1141e8ce, 0xa15486af, 0x7c72e993, 0xb3ee1411, 0x636fbc2a,
0x2ba9c55d, 0x741831f6, 0xce5c3e16, 0x9b87931e, 0xafd6ba33, 0x6c24cf5c,
0x7a325381, 0x28958677, 0x3b8f4898, 0x6b4bb9af, 0xc4bfe81b, 0x66282193,
0x61d809cc, 0xfb21a991, 0x487cac60, 0x5dec8032, 0xef845d5d, 0xe98575b1,
0xdc262302, 0xeb651b88, 0x23893e81, 0xd396acc5, 0x0f6d6ff3, 0x83f44239,
0x2e0b4482, 0xa4842004, 0x69c8f04a, 0x9e1f9b5e, 0x21c66842, 0xf6e96c9a,
0x670c9c61, 0xabd388f0, 0x6a51a0d2, 0xd8542f68, 0x960fa728, 0xab5133a3,
0x6eef0b6c, 0x137a3be4, 0xba3bf050, 0x7efb2a98, 0xa1f1651d, 0x39af0176,
0x66ca593e, 0x82430e88, 0x8cee8619, 0x456f9fb4, 0x7d84a5c3, 0x3b8b5ebe,
0xe06f75d8, 0x85c12073, 0x401a449f, 0x56c16aa6, 0x4ed3aa62, 0x363f7706,
0x1bfedf72, 0x429b023d, 0x37d0d724, 0xd00a1248, 0xdb0fead3, 0x49f1c09b,
0x075372c9, 0x80991b7b, 0x25d479d8, 0xf6e8def7, 0xe3fe501a, 0xb6794c3b,
0x976ce0bd, 0x04c006ba, 0xc1a94fb6, 0x409f60c4, 0x5e5c9ec2, 0x196a2463,
0x68fb6faf, 0x3e6c53b5, 0x1339b2eb, 0x3b52ec6f, 0x6dfc511f, 0x9b30952c,
0xcc814544, 0xaf5ebd09, 0xbee3d004, 0xde334afd, 0x660f2807, 0x192e4bb3,
0xc0cba857, 0x45c8740f, 0xd20b5f39, 0xb9d3fbdb, 0x5579c0bd, 0x1a60320a,
0xd6a100c6, 0x402c7279, 0x679f25fe, 0xfb1fa3cc, 0x8ea5e9f8, 0xdb3222f8,
0x3c7516df, 0xfd616b15, 0x2f501ec8, 0xad0552ab, 0x323db5fa, 0xfd238760,
0x53317b48, 0x3e00df82, 0x9e5c57bb, 0xca6f8ca0, 0x1a87562e, 0xdf1769db,
0xd542a8f6, 0x287effc3, 0xac6732c6, 0x8c4f5573, 0x695b27b0, 0xbbca58c8,
0xe1ffa35d, 0xb8f011a0, 0x10fa3d98, 0xfd2183b8, 0x4afcb56c, 0x2dd1d35b,
0x9a53e479, 0xb6f84565, 0xd28e49bc, 0x4bfb9790, 0xe1ddf2da, 0xa4cb7e33,
0x62fb1341, 0xcee4c6e8, 0xef20cada, 0x36774c01, 0xd07e9efe, 0x2bf11fb4,
0x95dbda4d, 0xae909198, 0xeaad8e71, 0x6b93d5a0, 0xd08ed1d0, 0xafc725e0,
0x8e3c5b2f, 0x8e7594b7, 0x8ff6e2fb, 0xf2122b64, 0x8888b812, 0x900df01c,
0x4fad5ea0, 0x688fc31c, 0xd1cff191, 0xb3a8c1ad, 0x2f2f2218, 0xbe0e1777,
0xea752dfe, 0x8b021fa1, 0xe5a0cc0f, 0xb56f74e8, 0x18acf3d6, 0xce89e299,
0xb4a84fe0, 0xfd13e0b7, 0x7cc43b81, 0xd2ada8d9, 0x165fa266, 0x80957705,
0x93cc7314, 0x211a1477, 0xe6ad2065, 0x77b5fa86, 0xc75442f5, 0xfb9d35cf,
0xebcdaf0c, 0x7b3e89a0, 0xd6411bd3, 0xae1e7e49, 0x00250e2d, 0x2071b35e,
0x226800bb, 0x57b8e0af, 0x2464369b, 0xf009b91e, 0x5563911d, 0x59dfa6aa,
0x78c14389, 0xd95a537f, 0x207d5ba2, 0x02e5b9c5, 0x83260376, 0x6295cfa9,
0x11c81968, 0x4e734a41, 0xb3472dca, 0x7b14a94a, 0x1b510052, 0x9a532915,
0xd60f573f, 0xbc9bc6e4, 0x2b60a476, 0x81e67400, 0x08ba6fb5, 0x571be91f,
0xf296ec6b, 0x2a0dd915, 0xb6636521, 0xe7b9f9b6, 0xff34052e, 0xc5855664,
0x53b02d5d, 0xa99f8fa1, 0x08ba4799, 0x6e85076a},
{0x4b7a70e9, 0xb5b32944,0xdb75092e, 0xc4192623, 0xad6ea6b0, 0x49a7df7d,
0x9cee60b8, 0x8fedb266, 0xecaa8c71, 0x699a17ff, 0x5664526c, 0xc2b19ee1,
0x193602a5, 0x75094c29, 0xa0591340, 0xe4183a3e, 0x3f54989a, 0x5b429d65,
0x6b8fe4d6, 0x99f73fd6, 0xa1d29c07, 0xefe830f5, 0x4d2d38e6, 0xf0255dc1,
0x4cdd2086, 0x8470eb26, 0x6382e9c6, 0x021ecc5e, 0x09686b3f, 0x3ebaefc9,
0x3c971814, 0x6b6a70a1, 0x687f3584, 0x52a0e286, 0xb79c5305, 0xaa500737,
0x3e07841c, 0x7fdeae5c, 0x8e7d44ec, 0x5716f2b8, 0xb03ada37, 0xf0500c0d,
```

```
        0xf01c1f04, 0x0200b3ff, 0xae0cf51a, 0x3cb574b2, 0x25837a58, 0xdc0921bd,
        0xd19113f9, 0x7ca92ff6, 0x94324773, 0x22f54701, 0x3ae5e581, 0x37c2dadc,
        0xc8b57634, 0x9af3dda7, 0xa9446146, 0x0fd0030e, 0xecc8c73e, 0xa4751e41,
        0xe238cd99, 0x3bea0e2f, 0x3280bba1, 0x183eb331, 0x4e548b38, 0x4f6db908,
        0x6f420d03, 0xf60a04bf, 0x2cb81290, 0x24977c79, 0x5679b072, 0xbcaf89af,
        0xde9a771f, 0xd9930810, 0xb38bae12, 0xdccf3f2e, 0x5512721f, 0x2e6b7124,
        0x501adde6, 0x9f84cd87, 0x7a584718, 0x7408da17, 0xbc9f9abc, 0xe94b7d8c,
        0xec7aec3a, 0xdb851dfa, 0x63094366, 0xc464c3d2, 0xef1c1847, 0x3215d908,
        0xdd433b37, 0x24c2ba16, 0x12a14d43, 0x2a65c451, 0x50940002, 0x133ae4dd,
        0x71dff89e, 0x10314e55, 0x81ac77d6, 0x5f11199b, 0x043556f1, 0xd7a3c76b,
        0x3c11183b, 0x5924a509, 0xf28fe6ed, 0x97f1fbfa, 0x9ebabf2c, 0x1e153c6e,
        0x86e34570, 0xeae96fb1, 0x860e5e0a, 0x5a3e2ab3, 0x771fe71c, 0x4e3d06fa,
        0x2965dcb9, 0x99e71d0f, 0x803e89d6, 0x5266c825, 0x2e4cc978, 0x9c10b36a,
        0xc6150eba, 0x94e2ea78, 0xa5fc3c53, 0x1e0a2df4, 0xf2f74ea7, 0x361d2b3d,
        0x1939260f, 0x19c27960, 0x5223a708, 0xf71312b6, 0xebadfe6e, 0xeac31f66,
        0xe3bc4595, 0xa67bc883, 0xb17f37d1, 0x018cff28, 0xc332ddef, 0xbe6c5aa5,
        0x65582185, 0x68ab9802, 0xeecea50f, 0xdb2f953b, 0x2aef7dad, 0x5b6e2f84,
        0x1521b628, 0x29076170, 0xecdd4775, 0x619f1510, 0x13cca830, 0xeb61bd96,
        0x0334fe1e, 0xaa0363cf, 0xb5735c90, 0x4c70a239, 0xd59e9e0b, 0xcbaade14,
        0xeecc86bc, 0x60622ca7, 0x9cab5cab, 0xb2f3846e, 0x648b1eaf, 0x19bdf0ca,
        0xa02369b9, 0x655abb50, 0x40685a32, 0x3c2ab4b3, 0x319ee9d5, 0xc021b8f7,
        0x9b540b19, 0x875fa099, 0x95f7997e, 0x623d7da8, 0xf837889a, 0x97e32d77,
        0x11ed935f, 0x16681281, 0x0e358829, 0xc7e61fd6, 0x96dedfa1, 0x7858ba99,
        0x57f584a5, 0x1b227263, 0x9b83c3ff, 0x1ac24696, 0xcdb30aeb, 0x532e3054,
        0x8fd948e4, 0x6dbc3128, 0x58ebf2ef, 0x34c6ffea, 0xfe28ed61, 0xee7c3c73,
        0x5d4a14d9, 0xe864b7e3, 0x42105d14, 0x203e13e0, 0x45eee2b6, 0xa3aaabea,
        0xdb6c4f15, 0xfacb4fd0, 0xc742f442, 0xef6abbb5, 0x654f3b1d, 0x41cd2105,
        0xd81e799e, 0x86854dc7, 0xe44b476a, 0x3d816250, 0xcf62a1f2, 0x5b8d2646,
        0xfc8883a0, 0xc1c7b6a3, 0x7f1524c3, 0x69cb7492, 0x47848a0b, 0x5692b285,
        0x095bbf00, 0xad19489d, 0x1462b174, 0x23820e00, 0x58428d2a, 0x0c55f5ea,
        0x1dadf43e, 0x233f7061, 0x3372f092, 0x8d937e41, 0xd65fecf1, 0x6c223bdb,
        0x7cde3759, 0xcbee7460, 0x4085f2a7, 0xce77326e, 0xa6078084, 0x19f8509e,
        0xe8efd855, 0x61d99735, 0xa969a7aa, 0xc50c06c2, 0x5a04abfc, 0x800bcadc,
        0x9e447a2e, 0xc3453484, 0xfdd56705, 0x0e1e9ec9, 0xdb73dbd3, 0x105588cd,
        0x675fda79, 0xe3674340, 0xc5c43465, 0x713e38d8, 0x3d28f89e, 0xf16dff20,
        0x153e21e7, 0x8fb03d4a, 0xe6e39f2b, 0xdb83adf7},
        {0xe93d5a68, 0x948140f7, 0xf64c261c, 0x94692934, 0x411520f7, 0x7602d4f7,
        0xbcf46b2e, 0xd4a20068, 0xd4082471, 0x3320f46a, 0x43b7d4b7, 0x500061af,
        0x1e39f62e, 0x97244546, 0x14214f74, 0xbf8b8840, 0x4d95fc1d, 0x96b591af,
        0x70f4ddd3, 0x66a02f45, 0xbfbc09ec, 0x03bd9785, 0x7fac6dd0, 0x31cb8504,
        0x96eb27b3, 0x55fd3941, 0xda2547e6, 0xabca0a9a, 0x28507825, 0x530429f4,
        0x0a2c86da, 0xe9b66dfb, 0x68dc1462, 0xd7486900, 0x680ec0a4, 0x27a18dee,
        0x4f3ffea2, 0xe887ad8c, 0xb58ce006, 0x7af4d6b6, 0xaace1e7c, 0xd3375fec,
        0xce78a399, 0x406b2a42, 0x20fe9e35, 0xd9f385b9, 0xee39d7ab, 0x3b124e8b,
       0x1dc9faf7, 0x4b6d1856, 0x26a36631, 0xeae397b2, 0x3a6efa74, 0xdd5b4332,
```

```
0x6841e7f7, 0xca7820fb, 0xfb0af54e, 0xd8feb397, 0x454056ac, 0xba489527,
0x55533a3a, 0x20838d87, 0xfe6ba9b7, 0xd096954b, 0x55a867bc, 0xa1159a58,
0xcca92963, 0x99e1db33, 0xa62a4a56, 0x3f3125f9, 0x5ef47e1c, 0x9029317c,
0xfdf8e802, 0x04272f70, 0x80bb155c, 0x05282ce3, 0x95c11548, 0xe4c66d22,
0x48c1133f, 0xc70f86dc, 0x07f9c9ee, 0x41041f0f, 0x404779a4, 0x5d886e17,
0x325f51eb, 0xd59bc0d1, 0xf2bcc18f, 0x41113564, 0x257b7834, 0x602a9c60,
0xdff8e8a3, 0x1f636c1b, 0x0e12b4c2, 0x02e1329e, 0xaf664fd1, 0xcad18115,
0x6b2395e0, 0x333e92e1, 0x3b240b62, 0xeebeb922, 0x85b2a20e, 0xe6ba0d99,
0xde720c8c, 0x2da2f728, 0xd0127845, 0x95b794fd, 0x647d0862, 0xe7ccf5f0,
0x5449a36f, 0x877d48fa, 0xc39dfd27, 0xf33e8d1e, 0x0a476341, 0x992eff74,
0x3a6f6eab, 0xf4f8fd37, 0xa812dc60, 0xa1ebddf8, 0x991be14c, 0xdb6e6b0d,
0xc67b5510, 0x6d672c37, 0x2765d43b, 0xdcd0e804, 0xf1290dc7, 0xcc00ffa3,
0xb5390f92, 0x690fed0b, 0x667b9ffb, 0xcedb7d9c, 0xa091cf0b, 0xd9155ea3,
0xbb132f88, 0x515bad24, 0x7b9479bf, 0x763bd6eb, 0x37392eb3, 0xcc115979,
0x8026e297, 0xf42e312d, 0x6842ada7, 0xc66a2b3b, 0x12754ccc, 0x782ef11c,
0x6a124237, 0xb79251e7, 0x06a1bbe6, 0x4bfb6350, 0x1a6b1018, 0x11caedfa,
0x3d25bdd8, 0xe2e1c3c9, 0x44421659, 0x0a121386, 0xd90cec6e, 0xd5abea2a,
0x64af674e, 0xda86a85f, 0xbebfe988, 0x64e4c3fe, 0x9dbc8057, 0xf0f7c086,
0x60787bf8, 0x6003604d, 0xd1fd8346, 0xf6381fb0, 0x7745ae04, 0xd736fccc,
0x83426b33, 0xf01eab71, 0xb0804187, 0x3c005e5f, 0x77a057be, 0xbde8ae24,
0x55464299, 0xbf582e61, 0x4e58f48f, 0xf2ddfda2, 0xf474ef38, 0x8789bdc2,
0x5366f9c3, 0xc8b38e74, 0xb475f255, 0x46fcd9b9, 0x7aeb2661, 0x8b1ddf84,
0x846a0e79, 0x915f95e2, 0x466e598e, 0x20b45770, 0x8cd55591, 0xc902de4c,
0xb90bace1, 0xbb8205d0, 0x11a86248, 0x7574a99e, 0xb77f19b6, 0xe0a9dc09,
0x662d09a1, 0xc4324633, 0xe85a1f02, 0x09f0be8c, 0x4a99a025, 0x1d6efe10,
0x1ab93d1d, 0x0ba5a4df, 0xa186f20f, 0x2868f169, 0xdcb7da83, 0x573906fe,
0xa1e2ce9b, 0x4fcd7f52, 0x50115e01, 0xa70683fa, 0xa002b5c4, 0x0de6d027,
0x9af88c27, 0x773f8641, 0xc3604c06, 0x61a806b5, 0xf0177a28, 0xc0f586e0,
0x006058aa, 0x30dc7d62, 0x11e69ed7, 0x2338ea63, 0x53c2dd94, 0xc2c21634,
0xbbcbee56, 0x90bcb6de, 0xebfc7da1, 0xce591d76, 0x6f05e409, 0x4b7c0188,
0x39720a3d, 0x7c927c24, 0x86e3725f, 0x724d9db9, 0x1ac15bb4, 0xd39eb8fc,
0xed545578, 0x08fca5b5, 0xd83d7cd3, 0x4dad0fc4, 0x1e50ef5e, 0xb161e6f8,
0xa28514d9, 0x6c51133c, 0x6fd5c7e7, 0x56e14ec4, 0x362abfce, 0xddc6c837,
0xd79a3234, 0x92638212, 0x670efa8e, 0x406000e0},
{0x3a39ce37, 0xd3faf5cf, 0xabc27737, 0x5ac52d1b, 0x5cb0679e, 0x4fa33742,
0xd3822740, 0x99bc9bbe, 0xd5118e9d, 0xbf0f7315, 0xd62d1c7e, 0xc700c47b,
0xb78c1b6b, 0x21a19045, 0xb26eb1be, 0x6a366eb4, 0x5748ab2f, 0xbc946e79,
0xc6a376d2, 0x6549c2c8, 0x530ff8ee, 0x468dde7d, 0xd5730a1d, 0x4cd04dc6,
0x2939bbdb, 0xa9ba4650, 0xac9526e8, 0xbe5ee304, 0xa1fad5f0, 0x6a2d519a,
0x63ef8ce2, 0x9a86ee22, 0xc089c2b8, 0x43242ef6, 0xa51e03aa, 0x9cf2d0a4,
0x83c061ba, 0x9be96a4d, 0x8fe51550, 0xba645bd6, 0x2826a2f9, 0xa73a3ae1,
0x4ba99586, 0xef5562e9, 0xc72fefd3, 0xf752f7da, 0x3f046f69, 0x77fa0a59,
0x80e4a915, 0x87b08601, 0x9b09e6ad, 0x3b3ee593, 0xe990fd5a, 0x9e34d797,
0x2cf0b7d9, 0x022b8b51, 0x96d5ac3a, 0x017da67d, 0xd1cf3ed6, 0x7c7d2d28,
0x1f9f25cf, 0xadf2b89b, 0x5ad6b472, 0x5a88f54c, 0xe029ac71, 0xe019a5e6,
```

```
0x47b0acfd, 0xed93fa9b, 0xe8d3c48d, 0x283b57cc, 0xf8d56629, 0x79132e28,
0x785f0191, 0xed756055, 0xf7960e44, 0xe3d35e8c, 0x15056dd4, 0x88f46dba,
0x03a16125, 0x0564f0bd, 0xc3eb9e15, 0x3c9057a2, 0x97271aec, 0xa93a072a,
0x1b3f6d9b, 0x1e6321f5, 0xf59c66fb, 0x26dcf319, 0x7533d928, 0xb155fdf5,
0x03563482, 0x8aba3cbb, 0x28517711, 0xc20ad9f8, 0xabcc5167, 0xccad925f,
0x4de81751, 0x3830dc8e, 0x379d5862, 0x9320f991, 0xea7a90c2, 0xfb3e7bce,
0x5121ce64, 0x774fbe32, 0xa8b6e37e, 0xc3293d46, 0x48de5369, 0x6413e680,
0xa2ae0810, 0xdd6db224, 0x69852dfd, 0x09072166, 0xb39a460a, 0x6445c0dd,
0x586cdecf, 0x1c20c8ae, 0x5bbef7dd, 0x1b588d40, 0xccd2017f, 0x6bb4e3bb,
0xdda26a7e, 0x3a59ff45, 0x3e350a44, 0xbcb4cdd5, 0x72eacea8, 0xfa6484bb,
0x8d6612ae, 0xbf3c6f47, 0xd29be463, 0x542f5d9e, 0xaec2771b, 0xf64e6370,
0x740e0d8d, 0xe75b1357, 0xf8721671, 0xaf537d5d, 0x4040cb08, 0x4eb4e2cc,
0x34d2466a, 0x0115af84, 0xe1b00428, 0x95983a1d, 0x06b89fb4, 0xce6ea048,
0x6f3f3b82, 0x3520ab82, 0x011a1d4b, 0x277227f8, 0x611560b1, 0xe7933fdc,
0xbb3a792b, 0x344525bd, 0xa08839e1, 0x51ce794b, 0x2f32c9b7, 0xa01fbac9,
0xe01cc87e, 0xbcc7d1f6, 0xcf0111c3, 0xa1e8aac7, 0x1a908749, 0xd44fbd9a,
0xd0dadecb, 0xd50ada38, 0x0339c32a, 0xc6913667, 0x8df9317c, 0xe0b12b4f,
0xf79e59b7, 0x43f5bb3a, 0xf2d519ff, 0x27d9459c, 0xbf97222c, 0x15e6fc2a,
0x0f91fc71, 0x9b941525, 0xfae59361, 0xceb69ceb, 0xc2a86459, 0x12baa8d1,
0xb6c1075e, 0xe3056a0c, 0x10d25065, 0xcb03a442, 0xe0ec6e0e, 0x1698db3b,
0x4c98a0be, 0x3278e964, 0x9f1f9532, 0xe0d392df, 0xd3a0342b, 0x8971f21e,
0x1b0a7441, 0x4ba3348c, 0xc5be7120, 0xc37632d8, 0xdf359f8d, 0x9b992f2e,
0xe60b6f47, 0x0fe3f11d, 0xe54cda54, 0x1edad891, 0xce6279cf, 0xcd3e7e6f,
0x1618b166, 0xfd2c1d05, 0x848fd2c5, 0xf6fb2299, 0xf523f357, 0xa6327623,
0x93a83531, 0x56cccd02, 0xacf08162, 0x5a75ebb5, 0x6e163697, 0x88d273cc,
0xde966292, 0x81b949d0, 0x4c50901b, 0x71c65614, 0xe6c6c7bd, 0x327a140a,
0x45e1d006, 0xc3f27b9a, 0xc9aa53fd, 0x62a80f00, 0xbb25bfe2, 0x35bdd2f6,
0x71126905, 0xb2040222, 0xb6cbcf7c, 0xcd769c2b, 0x53113ec0, 0x1640e3d3,
0x38abbd60, 0x2547adf0, 0xba38209c, 0xf746ce76, 0x77afa1c5, 0x20756060,
0x85cbfe4e, 0x8ae88dd8, 0x7aaaf9b0, 0x4cf9aa7e, 0x1948c25c, 0x02fb8a8c,
0x01c36ae4, 0xd6ebe1f9, 0x90d4f869, 0xa65cdea0, 0x3f09252d, 0xc208e69f,
0xb74e6132, 0xce77e25b, 0x578fdfe3, 0x3ac372e6}};
```

根据加密密钥生成子密钥 P 和 S 盒的具体过程见程序清单 7-6。

程序清单 7-6

```
void Blowfish::init(unsigned char * key,int keyLength)
{
    int i,j,k;
    word32 data,dataL,dataR;
    for(i=0;i<4;i++)
    {
        for(j=0;j<256;j++)
        {
            sBox[i][j]=originSBox[i][j];
        }
```

```
11          }
12          j=0;
13          for(i=0;i<18;i++)
14          {
15                data=0x00000000;
16                for(k=0;k<4;++k)
17                {
18                      data=(data<<8)|key[j];
19                      j=j+1;
20                      if(j>=keyLength)
21                      {
22                            j=0;
23                      }
24                }
25                P[i]=originP[i]^data;
26          }
27          dataL=0x00000000;
28          dataR=0x00000000;
29          for(i=0;i<18;i+=2)
30          {
31                encryption(&dataL,&dataR);
32                P[i]=dataL;
33                P[i+1]=dataR;
34          }
35          for(i=0;i<4;++i)
36          {
37                for(j=0;j<256;j+=2)
38                {
39                      encryption(&dataL,&dataR);
40                      sBox[i][j]=dataL;
41                      sBox[i][j+1]=dataR;
42                }
43          }
44    }
```

初始化函数 init()的参数为输入密钥和密钥长度。代码行第 5 行到第 11 行是初始化 S 盒,将常量 S 盒的值赋给具体计算用的 S 盒,代码行第 13 行到第 34 行是计算实际加密和解密用的子密钥 P,第 18 行代码是按字节将输入的密钥填充到 data,data 是 32 位数据,每次填充 8 位,填充的方法是采用"|"运算,在填充完一个字节之后,将 data 右移 8 位,再进行下一个字节的填充,直到 32 位数据全部填充完毕。在计算过程中输入的初始密钥是循环使用的,第 20 行到第 23 行代码用于处理输入密钥的循环使用,当使用了最后一个密钥后,又从第一个输入密钥开始重复使用输入密钥。第 29 行到第 34 行代码是子密钥计算过程的最后一步,在这步计算中得到的 dataL 和 dataR 将用于 S 盒的计算。代码行第 35 行到第 43 行是计算实际加密或解密用的 S 盒。整个过程完成计算实际加密和解密用的子密钥 P 和

S 盒。

7.2.3 程序测试

程序测试通过主函数驱动测试函数 Test()来完成,测试函数 Test()的代码见程序清单 7-7。

程序清单 7-7

```
void Blowfish::Test()
{
    ofstream out("test.out");
    word32 L=23,R=43;
    out<<"The PlainText are:"<<endl;
    out<<"L="<<hex<<L<<endl;
    out<<"R="<<hex<<R<<endl;
    encryption(&L,&R);
    out<<"The enCipherText are:"<<endl;
    out<<"L="<<hex<<L<<endl;
    out<<"R="<<hex<<R<<endl;
    decryption(&L,&R);
    out<<"The deCipherText are:"<<endl;
    out<<"L="<<hex<<L<<endl;
    out<<"R="<<hex<<R<<endl;
}
```

测试函数主要完成输入明文、加密、输出密文、解密、输出解密后的数据,测试得到的结果输出到文件“test. out”,在本测试用例中只是对两个 32 位的输入数据进行简单的测试,在具体使用中可以通过从文件中读取数据,然后再进行加密和解密的方式进行处理。

测试函数通过主函数驱动,主函数见程序清单 7-8。

程序清单 7-8

```
int main()
{
    Blowfish bf;
    bf.init((unsigned char*)"TwoFish",7);
    bf.Test();
    return 0;
}
```

主函数主要完成构造 Blowfish 类的对象、调用初始化函数并传递密钥和密钥长度、调用测试函数。测试结果如下:

```
The PlainText are:
L=17
R=2b
The enCipherText are:
```

```
L= 3032da2c
R= 3480d712
The deCipherText are:
L= 17
R= 2b
```

经过解密后的数据与输入的明文一致,该程序测试的数据时直接将明文分为左右各半进行处理,即加密的数据是两个 32 位的 word32 型数据,在输出数据时使用的是十六进制格式的数据,若直接输入 64 位数据或输入字符只需要进行相应的转换即可。

7.3 习题与实践题

7.3.1 习题

1. 简要说明 Blowfish 加密算法的加密和解密的基本过程。
2. 简要说明 Blowfish 加密算法中的 F 函数运算的基本原理,并用示例说明。
3. 简要说明 Blowfish 加密算法的密钥生成过程。

7.3.2 实践题

参考 7.2 节的 Blowfish 算法的实现过程,完成 Blowfish 加密算法,要求:待加密消息从文件中读取,加密后的消息存储到对应的文件,解密后的消息存储到相应的另一个文件。

第 8 章 CAST-128 算法

CAST 算法是一系列算法，目前主要有 CAST-128 和 CAST-256 两种算法。CAST 算法是由加拿大 Queen 大学的 Carlisle Adams 和 Stanford Tavares 设计的，因此该算法以 CAST 命名，后期的改进工作主要由 Carlisle Adams 和 Michael Wiener 来完成。

CAST 算法也是 Feistel 结构的分组密码算法，对于微分密码分析和线性密码分析等具有较好的抵抗性。

8.1 CAST-128 算法原理

CAST-128 加密算法的输入是 64 位明文，加密使用的密钥长度通常为 128 位，输出的密文为 64 位。CAST-128 加密算法的基本过程如图 8-1 所示。

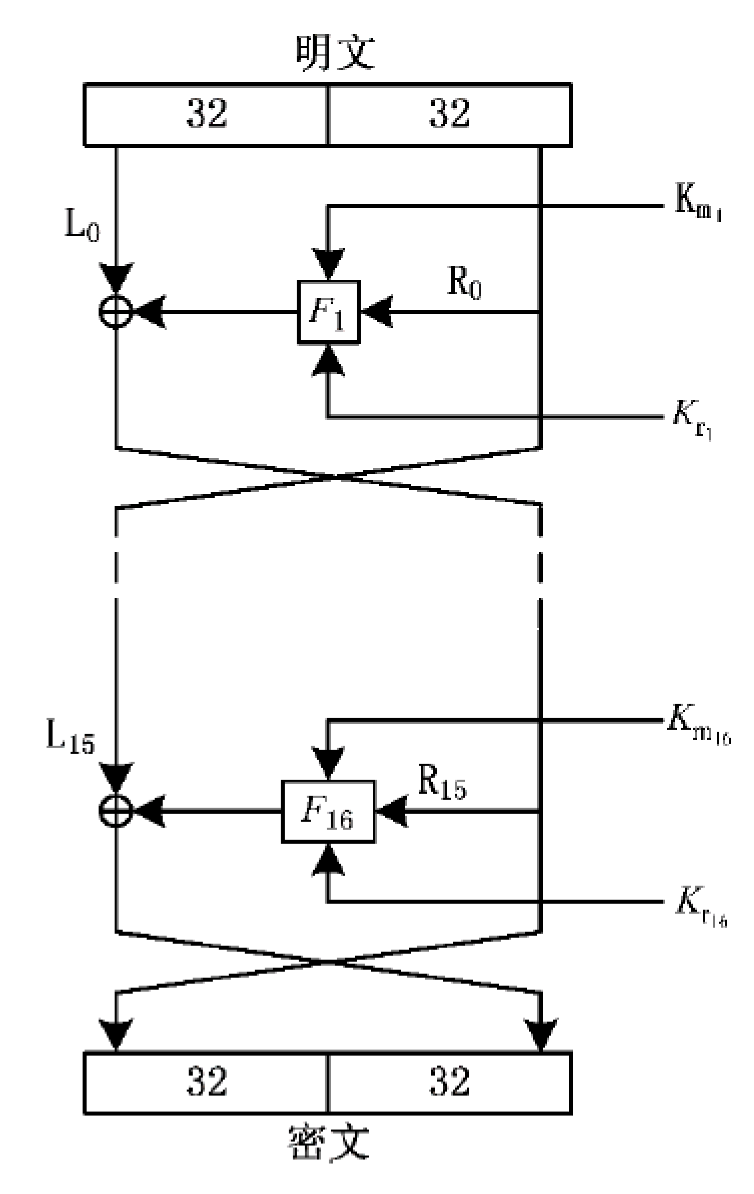


图 8-1 CAST-128 加密算法基本过程

在 CAST-128 算法中用到的$\{K_{m_i}, K_{r_i}\}$是轮密钥，轮密钥是通过输入的密钥计算获得。通常 CAST 加密的轮数为 16 轮。

8.1.1 CAST-128 算法的加密过程

假设输入的 64 位明文用 $m_1m_2\cdots m_{64}$ 表示，输入的密钥用 $K=k_1k_2\cdots k_{128}$ 表示，输出的密文用 $c_1c_2\cdots c_{64}$ 表示，CAST-128 加密算法的加密过程可以用以下过程描述：

(1) 根据输入的密钥计算子密钥对$\{K_{m_i}, K_{r_i}\}$，子密钥共 16 对，分别用于 16 轮加密运算。其中：K_{m_i} 为掩码子密钥，K_{r_i} 为旋转子密钥。

(2) 将输入的明文分为左右两半为 L_0 和 R_0，L_0 为输入明文的高 32 位，即 $L_0=m_1\cdots m_{32}$，R_0 为输入明文的低 32 位，即 $R_0=m_{33}\cdots m_{64}$。

(3) 进行 16 轮加密运算，每一轮的加密运算的过程见公式 8-1：

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \hat{} f(R_{i-1}, K_{m_i}, K_{r_i}) \end{aligned} \tag{8-1}$$

(4) 将最后一轮得到的数据进行左右交换，得到(R_{16}, L_{16})，即 $c_1c_2\cdots c_{64}$。

CAST-128 的解密过程与加密过程一样，但使用子密钥的顺序正好相反。

在 16 轮的加密运算过程中用到了 F 函数，F 函数的执行过程如图 8-2 所示。

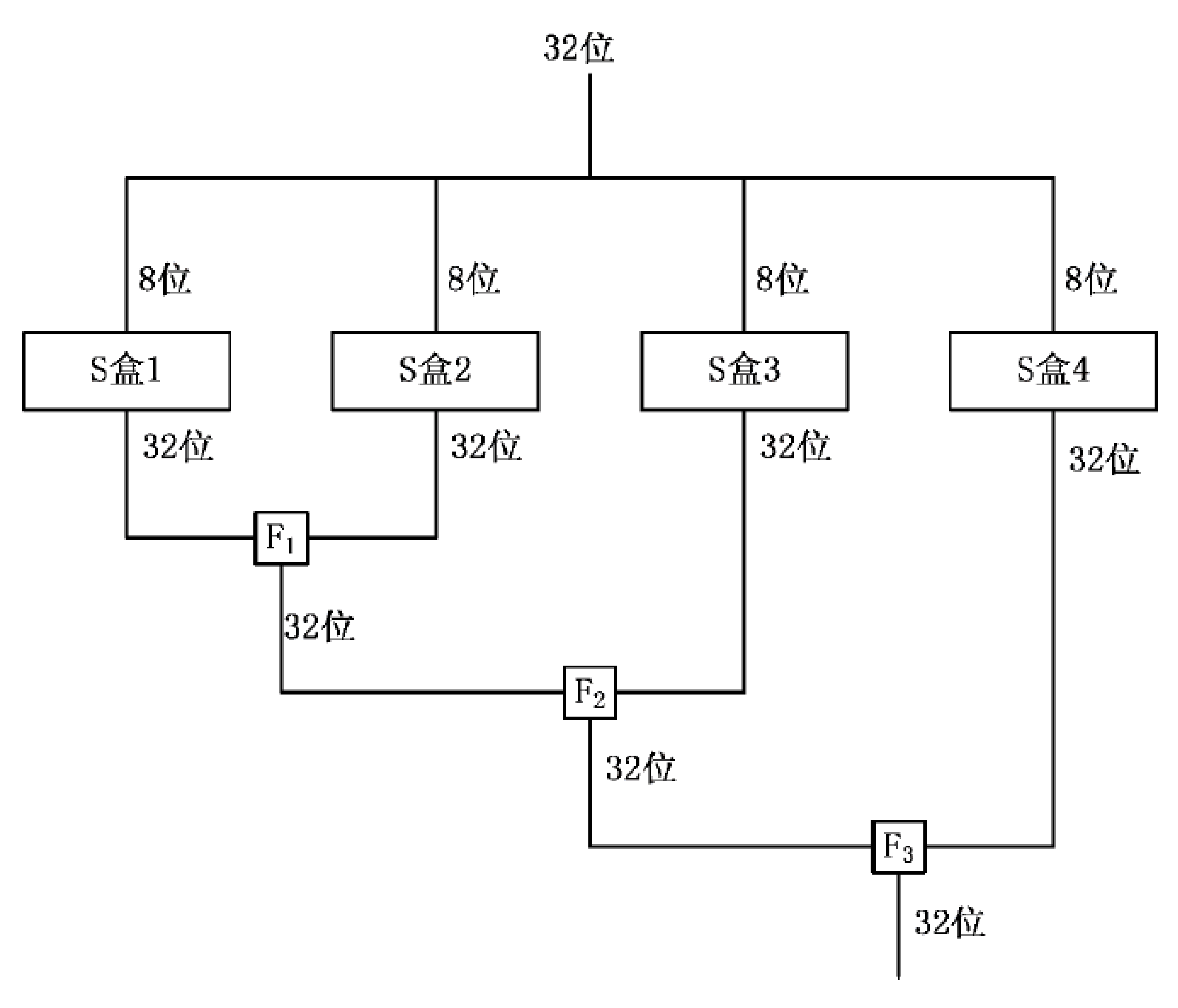


图 8-2 F 函数运算过程示意图

CAST-128 加密算法的 F 函数根据加密轮次的不同而不同。设 F 函数的输入数据为 D，“I_a”、“I_b”、“I_c”、“I_d”为数据 I 的各个字节，“<<<”表示循环左移，那么有三种 F 函数如下：

$$
\begin{aligned}
&\text{类型 1：}\begin{cases} I = ((\mathrm{K}_{\mathrm{m}_i} + D) <<< \mathrm{K}_{\mathrm{r}_i}) \\ f = ((S_1[I_a] \wedge S_2[I_b]) - S_3[I_c] + S_4[I_d]) \end{cases} \\
&\text{类型 2：}\begin{cases} I = ((\mathrm{K}_{\mathrm{m}_i} \wedge D) <<< \mathrm{K}_{\mathrm{r}_i}) \\ f = ((S_1[I_a] - S_2[I_b]) + S_3[I_c]) \wedge S_4[I_d] \end{cases} \\
&\text{类型 3：}\begin{cases} I = ((\mathrm{K}_{\mathrm{m}_i} - D) <<< \mathrm{K}_{\mathrm{r}_i}) \\ f = (S_1[I_a] + S_2[I_b]) \wedge S_3[I_c] - S_4[I_d] \end{cases}
\end{aligned} \tag{8-2}
$$

对于第 1、4、7、10、13 和 16 轮加密，F 函数采用第 1 种运算。

对于第 2、5、8、11 和 14 轮加密，F 函数采用第 2 种运算。

对于第 3、6、9、12 和 15 轮加密，F 函数采用第 3 种运算。

CAST-128 算法中共有 8 个 S 盒，每个 S 盒的大小为 256，其中 4 个 S 盒用于加密算法，另 4 个 S 盒用于密钥生成。

CAST-128 算法的解密过程与加密过程一样，只是密钥的使用顺序与加密过程中密钥的使用顺序相反。

8.1.2 CAST-128 算法的子密钥生成

CAST-128 算法的输入密钥为 128 位，若以字节的形式表示，那么输入的密钥可以表示为：x0x1x2x3x4x5x6x7x8x9xAxBxCxDxExF，共 16 个字节。

CAST-128 算法的密钥由两部分组成，一部分为“掩码”密钥 K_m，另一部分为“旋转”密钥 K_r，每一轮的加密运算使用一组密钥，即一个“掩码”密钥和一个“旋转”密钥，因此 16 轮运算共需要 32 个密钥。“旋转”密钥实际上就使用密钥的最后 5 位。

设 z0z1z2z3z4z5z6z7z8z9zAzBzCzDzEzF 为计算子密钥用的 16 个字节的临时变量，那么密钥的计算过程如下：

```
z0z1z2z3=x0x1x2x3 ^ S5[xD] ^ S6[xF] ^ S7[xC] ^ S8[xE] ^ S7[x8]
z4z5z6z7=x8x9xAxB ^ S5[z0] ^ S6[z2] ^ S7[z1] ^ S8[z3] ^ S8[xA]
z8z9zAzB=xCxDxExF ^ S5[z7] ^ S6[z6] ^ S7[z5] ^ S8[z4] ^ S5[x9]
zCzDzEzF=x4x5x6x7 ^ S5[zA] ^ S6[z9] ^ S7[zB] ^ S8[z8] ^ S6[xB]
K1=S5[z8] ^ S6[z9] ^ S7[z7] ^ S8[z6] ^ S5[z2]
K2=S5[zA] ^ S6[zB] ^ S7[z5] ^ S8[z4] ^ S6[z6]
K3=S5[zC] ^ S6[zD] ^ S7[z3] ^ S8[z2] ^ S7[z9]
K4=S5[zE] ^ S6[zF] ^ S7[z1] ^ S8[z0] ^ S8[zC]
x0x1x2x3=z8z9zAzB ^ S5[z5] ^ S6[z7] ^ S7[z4] ^ S8[z6] ^ S7[z0]
x4x5x6x7=z0z1z2z3 ^ S5[x0] ^ S6[x2] ^ S7[x1] ^ S8[x3] ^ S8[z2]
x8x9xAxB=z4z5z6z7 ^ S5[x7] ^ S6[x6] ^ S7[x5] ^ S8[x4] ^ S5[z1]
xCxDxExF=zCzDzEzF ^ S5[xA] ^ S6[x9] ^ S7[xB] ^ S8[x8] ^ S6[z3]
K5=S5[x3] ^ S6[x2] ^ S7[xC] ^ S8[xD] ^ S5[x8]
K6=S5[x1] ^ S6[x0] ^ S7[xE] ^ S8[xF] ^ S6[xD]
K7=S5[x7] ^ S6[x6] ^ S7[x8] ^ S8[x9] ^ S7[x3]
K8=S5[x5] ^ S6[x4] ^ S7[xA] ^ S8[xB] ^ S8[x7]
z0z1z2z3=x0x1x2x3 ^ S5[xD] ^ S6[xF] ^ S7[xC] ^ S8[xE] ^ S7[x8]
z4z5z6z7=x8x9xAxB ^ S5[z0] ^ S6[z2] ^ S7[z1] ^ S8[z3] ^ S8[xA]
z8z9zAzB=xCxDxExF ^ S5[z7] ^ S6[z6] ^ S7[z5] ^ S8[z4] ^ S5[x9]
zCzDzEzF=x4x5x6x7 ^ S5[zA] ^ S6[z9] ^ S7[zB] ^ S8[z8] ^ S6[xB]
K9=S5[z3] ^ S6[z2] ^ S7[zC] ^ S8[zD] ^ S5[z9]
K10=S5[z1] ^ S6[z0] ^ S7[zE] ^ S8[zF] ^ S6[zC]
K11=S5[z7] ^ S6[z6] ^ S7[z8] ^ S8[z9] ^ S7[z2]
K12=S5[z5] ^ S6[z4] ^ S7[zA] ^ S8[zB] ^ S8[z6]
x0x1x2x3=z8z9zAzB ^ S5[z5] ^ S6[z7] ^ S7[z4] ^ S8[z6] ^ S7[z0]
x4x5x6x7=z0z1z2z3 ^ S5[x0] ^ S6[x2] ^ S7[x1] ^ S8[x3] ^ S8[z2]
x8x9xAxB=z4z5z6z7 ^ S5[x7] ^ S6[x6] ^ S7[x5] ^ S8[x4] ^ S5[z1]
xCxDxExF=zCzDzEzF ^ S5[xA] ^ S6[x9] ^ S7[xB] ^ S8[x8] ^ S6[z3]
K13=S5[x8] ^ S6[x9] ^ S7[x7] ^ S8[x6] ^ S5[x3]
K14=S5[xA] ^ S6[xB] ^ S7[x5] ^ S8[x4] ^ S6[x7]
K15=S5[xC] ^ S6[xD] ^ S7[x3] ^ S8[x2] ^ S7[x8]
K16=S5[xE] ^ S6[xF] ^ S7[x1] ^ S8[x0] ^ S8[xD]
z0z1z2z3=x0x1x2x3 ^ S5[xD] ^ S6[xF] ^ S7[xC] ^ S8[xE] ^ S7[x8]
z4z5z6z7=x8x9xAxB ^ S5[z0] ^ S6[z2] ^ S7[z1] ^ S8[z3] ^ S8[xA]
z8z9zAzB=xCxDxExF ^ S5[z7] ^ S6[z6] ^ S7[z5] ^ S8[z4] ^ S5[x9]
zCzDzEzF=x4x5x6x7 ^ S5[zA] ^ S6[z9] ^ S7[zB] ^ S8[z8] ^ S6[xB]
K17=S5[z8] ^ S6[z9] ^ S7[z7] ^ S8[z6] ^ S5[z2]
K18=S5[zA] ^ S6[zB] ^ S7[z5] ^ S8[z4] ^ S6[z6]
K19=S5[zC] ^ S6[zD] ^ S7[z3] ^ S8[z2] ^ S7[z9]
```

```
K20= S5[zE] ^ S6[zF] ^ S7[z1] ^ S8[z0] ^ S8[zC]
x0x1x2x3= z8z9zAzB ^ S5[z5] ^ S6[z7] ^ S7[z4] ^ S8[z6] ^ S7[z0]
x4x5x6x7= z0z1z2z3 ^ S5[x0] ^ S6[x2] ^ S7[x1] ^ S8[x3] ^ S8[z2]
x8x9xAxB= z4z5z6z7 ^ S5[x7] ^ S6[x6] ^ S7[x5] ^ S8[x4] ^ S5[z1]
xCxDxExF= zCzDzEzF ^ S5[xA] ^ S6[x9] ^ S7[xB] ^ S8[x8] ^ S6[z3]
K21= S5[x3] ^ S6[x2] ^ S7[xC] ^ S8[xD] ^ S5[x8]
K22= S5[x1] ^ S6[x0] ^ S7[xE] ^ S8[xF] ^ S6[xD]
K23= S5[x7] ^ S6[x6] ^ S7[x8] ^ S8[x9] ^ S7[x3]
K24= S5[x5] ^ S6[x4] ^ S7[xA] ^ S8[xB] ^ S8[x7]
z0z1z2z3= x0x1x2x3 ^ S5[xD] ^ S6[xF] ^ S7[xC] ^ S8[xE] ^ S7[x8]
z4z5z6z7= x8x9xAxB ^ S5[z0] ^ S6[z2] ^ S7[z1] ^ S8[z3] ^ S8[xA]
z8z9zAzB= xCxDxExF ^ S5[z7] ^ S6[z6] ^ S7[z5] ^ S8[z4] ^ S5[x9]
zCzDzEzF= x4x5x6x7 ^ S5[zA] ^ S6[z9] ^ S7[zB] ^ S8[z8] ^ S6[xB]
K25= S5[z3] ^ S6[z2] ^ S7[zC] ^ S8[zD] ^ S5[z9]
K26= S5[z1] ^ S6[z0] ^ S7[zE] ^ S8[zF] ^ S6[zC]
K27= S5[z7] ^ S6[z6] ^ S7[z8] ^ S8[z9] ^ S7[z2]
K28= S5[z5] ^ S6[z4] ^ S7[zA] ^ S8[zB] ^ S8[z6]
x0x1x2x3= z8z9zAzB ^ S5[z5] ^ S6[z7] ^ S7[z4] ^ S8[z6] ^ S7[z0]
x4x5x6x7= z0z1z2z3 ^ S5[x0] ^ S6[x2] ^ S7[x1] ^ S8[x3] ^ S8[z2]
x8x9xAxB= z4z5z6z7 ^ S5[x7] ^ S6[x6] ^ S7[x5] ^ S8[x4] ^ S5[z1]
xCxDxExF= zCzDzEzF ^ S5[xA] ^ S6[x9] ^ S7[xB] ^ S8[x8] ^ S6[z3]
K29= S5[x8] ^ S6[x9] ^ S7[x7] ^ S8[x6] ^ S5[x3]
K30= S5[xA] ^ S6[xB] ^ S7[x5] ^ S8[x4] ^ S6[x7]
K31= S5[xC] ^ S6[xD] ^ S7[x3] ^ S8[x2] ^ S7[x8]
K32= S5[xE] ^ S6[xF] ^ S7[x1] ^ S8[x0] ^ S8[xD]
```

将获得的 32 个密钥分成两部分，K1～K16 为密钥 K_{m_i}，K17～K32 为 K_{r_i}。

8.2 CAST-128 算法实现

CAST-128 算法的实现主要包含密钥生成、加密和解密以及数据的初始化，密钥的生成和加密的过程分别使用了不同的 S 盒。在加密过程中根据加密轮次的不同使用了不同 *F* 函数。

CAST-128 算法的实现主要有以下过程：

(1) 初始化 S 盒和密钥。

(2) 计算“掩码”密钥和“旋转”密钥。

(3) 进行加密和解密。

在程序处理过程中用到了 64 位数据、32 位数据和 8 位数据，通过相关声明来处理不同长度的数据以方便后续程序的使用，自定义数据类型包括：

```
typedef unsigned char byte;
typedef unsigned long word32;
typedef unsigned long long word64;
```

这些数据类型分别用于处理对应长度的数据。

CAST-128 算法实现的主要类的结构如图 8-3 所示。

CAST_128	
- x[16]	: byte
- roundKey[32]	: word32
- keyM[16]	: word32
- keyR[16]	: word32
- S[8][256]	: static const word32
- word32ToByte (word32 in, byte bIn[], int position)	: void
+ getKey (byte keyIn[])	: void
+ F (word32 in, word32 km, word32 kr, int round)	: word32
+ encryprion (word64 in)	: word64
+ decryption (word64 in)	: word64

图 8-3　CAST_128 类的基本结构

CAST_128 类的具体声明见程序清单 8-1。

程序清单 8-1

```
class CAST_128
{
    public:
        void getKey(byte keyIn[]);
        word64 encryption(word64 in);
        word64 decryption(word64 in);
        word32 F(word32 in,word32 km,word32 kr,int round);
    private:
        void word32ToByte(word32 in,byte bIn[],int position);
        byte x[16];
        word32 roundKey[32];
        word32 keyM[16];
        word32 keyR[16];
        static const word32 S[8][256];
};
```

在 CAST_128 类中：字节型变量 x[16]用于存储输入的密钥，word32 型变量 roundKey[32]用于存储计算获得的轮密钥，word32 型变量 keyM[16]用于存储计算获得的“掩码”密钥，word32 型变量 keyR[16]用于存储计算获得的“旋转”密钥，word32 型变量 S[8][256]用于存储 S 盒数据。函数 getKey()用于获得输入的密钥，并计算加密密钥和解密密钥，函数 encryption()用于加密输入的数据，函数 decryption()用于解密密文，函数 F()为每一轮计算过程中的 *F* 函数，函数 word32ToByte()为将 word32 型数据转换为 4 个 byte 型数据。

8.2.1　密钥生成

CAST_128 类中的密钥生成通过函数 getKey()来实现，生成的过程由三部分组成：获取输入密钥、计算轮密钥、计算“掩码”密钥和“旋转”密钥，密钥生成的具体代码见程序清

单 8-2。

程序清单 8-2

```
void CAST_128::getKey(byte keyIn[])
{
    int i;
    for(i=0;i<16;i++)
    {
        x[i]=keyIn[i];
    }
    word32 keyX[4];          //用于计算密钥的 word32 型数据
    word32 keyZ[4];          //用于计算密钥的 word32 型临时数据
    byte z[16];              //用于 S 盒的 keyZ 分量
    //生成 32 位的输入密钥组,共 4 个
    for(i=0;i<4;i++)
    {
        keyX[i]=(x[i*4]<<24)|(x[i*4+1]<<16)|(x[i*4+2]<<8)|(x[i*4+3]);
    }
    //第 1 轮密钥计算
    keyZ[0]=keyX[0]^S[4][x[13]]^S[5][x[15]]^S[6][x[12]]^S[7][x[14]]^S[6][x[8]];
    word32ToByte(keyZ[0],z,0);
    keyZ[1]=keyX[2]^S[4][z[0]]^S[5][z[2]]^S[6][z[1]]^S[7][z[3]]^S[7][x[10]];
    word32ToByte(keyZ[1],z,1);
    keyZ[2]=keyX[3]^S[4][z[7]]^S[5][z[6]]^S[6][z[5]]^S[7][z[4]]^S[4][x[9]];
    word32ToByte(keyZ[2],z,2);
    keyZ[3]=keyX[1]^S[4][z[10]]^S[5][z[9]]^S[6][z[11]]^S[7][z[8]]^S[5][x[11]];
    word32ToByte(keyZ[3],z,3);
    roundKey[0]=S[4][z[8]]^S[5][z[9]]^S[6][z[7]]^S[7][z[6]]^S[4][z[2]];
    roundKey[1]=S[4][z[10]]^S[5][z[11]]^S[6][z[5]]^S[7][z[4]]^S[5][z[6]];
    roundKey[2]=S[4][z[12]]^S[5][z[13]]^S[6][z[3]]^S[7][z[2]]^S[6][z[9]];
    roundKey[3]=S[4][z[14]]^S[5][z[15]]^S[6][z[1]]^S[7][z[0]]^S[7][z[12]];
    //第 2 轮密钥计算
    keyX[0]=keyZ[2]^S[4][z[5]]^S[5][z[7]]^S[6][z[4]]^S[7][z[6]]^S[6][z[0]];
    word32ToByte(keyX[0],x,0);
    keyX[1]=keyZ[0]^S[4][x[0]]^S[5][x[2]]^S[6][x[1]]^S[7][x[3]]^S[7][z[2]];
    word32ToByte(keyX[1],x,1);
    keyX[2]=keyZ[1]^S[4][x[7]]^S[5][x[6]]^S[6][x[5]]^S[7][x[4]]^S[4][z[1]];
    word32ToByte(keyX[2],x,2);
    keyX[3]=keyZ[3]^S[4][x[10]]^S[5][x[9]]^S[6][x[11]]^S[7][x[8]]^S[5][z[3]];
    word32ToByte(keyX[3],x,3);
    roundKey[4]=S[4][x[3]]^S[5][x[2]]^S[6][x[12]]^S[7][x[13]]^S[4][x[8]];
    roundKey[5]=S[4][x[1]]^S[5][x[0]]^S[6][x[14]]^S[7][x[15]]^S[5][x[13]];
    roundKey[6]=S[4][x[7]]^S[5][x[6]]^S[6][x[8]]^S[7][x[9]]^S[6][x[3]];
    roundKey[7]=S[4][x[5]]^S[5][x[4]]^S[6][x[10]]^S[7][x[11]]^S[7][x[7]];
    //第 3 轮密钥计算
```

```
keyZ[0]=keyX[0]^S[4][x[13]]^S[5][x[15]]^S[6][x[12]]^S[7][x[14]]^S[6][x[8]];
word32ToByte(keyZ[0],z,0);
keyZ[1]=keyX[2]^S[4][z[0]]^S[5][z[2]]^S[6][z[1]]^S[7][z[3]]^S[7][x[10]];
word32ToByte(keyZ[1],z,1);
keyZ[2]=keyX[3]^S[4][z[7]]^S[5][z[6]]^S[6][z[5]]^S[7][z[4]]^S[4][x[9]];
word32ToByte(keyZ[2],z,2);
keyZ[3]=keyX[1]^S[4][z[10]]^S[5][z[9]]^S[6][z[11]]^S[7][z[8]]^S[5][x[11]];
word32ToByte(keyZ[3],z,3);
roundKey[8]=S[4][z[3]]^S[5][z[2]]^S[6][z[12]]^S[7][z[13]]^S[4][z[9]];
roundKey[9]=S[4][z[1]]^S[5][z[0]]^S[6][z[14]]^S[7][z[15]]^S[5][z[12]];
roundKey[10]=S[4][z[7]]^S[5][z[6]]^S[6][z[8]]^S[7][z[9]]^S[6][z[2]];
roundKey[11]=S[4][z[5]]^S[5][z[4]]^S[6][z[10]]^S[7][z[11]]^S[7][z[6]];
//第 4 轮密钥计算
keyX[0]=keyZ[2]^S[4][z[5]]^S[5][z[7]]^S[6][z[4]]^S[7][z[6]]^S[6][z[0]];
word32ToByte(keyX[0],x,0);
keyX[1]=keyZ[0]^S[4][x[0]]^S[5][x[2]]^S[6][x[1]]^S[7][x[3]]^S[7][z[2]];
word32ToByte(keyX[1],x,1);
keyX[2]=keyZ[1]^S[4][x[7]]^S[5][x[6]]^S[6][x[5]]^S[7][x[4]]^S[4][z[1]];
word32ToByte(keyX[2],x,2);
keyX[3]=keyZ[3]^S[4][x[10]]^S[5][x[9]]^S[6][x[11]]^S[7][x[8]]^S[5][z[3]];
word32ToByte(keyX[3],x,3);
roundKey[12]=S[4][x[8]]^S[5][x[9]]^S[6][x[7]]^S[7][x[6]]^S[4][x[3]];
roundKey[13]=S[4][x[10]]^S[5][x[11]]^S[6][x[5]]^S[7][x[4]]^S[5][x[7]];
roundKey[14]=S[4][x[12]]^S[5][x[13]]^S[6][x[3]]^S[7][x[2]]^S[6][x[8]];
roundKey[15]=S[4][x[14]]^S[5][x[15]]^S[6][x[1]]^S[7][x[0]]^S[7][x[13]];
//第 5 轮密钥计算
keyZ[0]=keyX[0]^S[4][x[13]]^S[5][x[15]]^S[6][x[12]]^S[7][x[14]]^S[6][x[8]];
word32ToByte(keyZ[0],z,0);
keyZ[1]=keyX[2]^S[4][z[0]]^S[5][z[2]]^S[6][z[1]]^S[7][z[3]]^S[7][x[10]];
word32ToByte(keyZ[1],z,1);
keyZ[2]=keyX[3]^S[4][z[7]]^S[5][z[6]]^S[6][z[5]]^S[7][z[4]]^S[4][x[9]];
word32ToByte(keyZ[2],z,2);
keyZ[3]=keyX[1]^S[4][z[10]]^S[5][z[9]]^S[6][z[11]]^S[7][z[8]]^S[5][x[11]];
word32ToByte(keyZ[3],z,3);
roundKey[16]=S[4][z[8]]^S[5][z[9]]^S[6][z[7]]^S[7][z[6]]^S[4][z[2]];
roundKey[17]=S[4][z[10]]^S[5][z[11]]^S[6][z[5]]^S[7][z[4]]^S[5][z[6]];
roundKey[18]=S[4][z[12]]^S[5][z[13]]^S[6][z[3]]^S[7][z[2]]^S[6][z[9]];
roundKey[19]=S[4][z[14]]^S[5][z[15]]^S[6][z[1]]^S[7][z[0]]^S[7][z[12]];
//第 6 轮密钥计算
keyX[0]=keyZ[2]^S[4][z[5]]^S[5][z[7]]^S[6][z[4]]^S[7][z[6]]^S[6][z[0]];
word32ToByte(keyX[0],x,0);
keyX[1]=keyZ[0]^S[4][x[0]]^S[5][x[2]]^S[6][x[1]]^S[7][x[3]]^S[7][z[2]];
word32ToByte(keyX[1],x,1);
keyX[2]=keyZ[1]^S[4][x[7]]^S[5][x[6]]^S[6][x[5]]^S[7][x[4]]^S[4][z[1]];
word32ToByte(keyX[2],x,2);
```

```
88    keyX[3]=keyZ[3]^S[4][x[10]]^S[5][x[9]]^S[6][x[11]]^S[7][x[8]]^S[5][z[3]];
89    word32ToByte(keyX[3],x,3);
90    roundKey[20]=S[4][x[3]]^S[5][x[2]]^S[6][x[12]]^S[7][x[13]]^S[4][x[8]];
91    roundKey[21]=S[4][x[1]]^S[5][x[0]]^S[6][x[14]]^S[7][x[15]]^S[5][x[13]];
92    roundKey[22]=S[4][x[7]]^S[5][x[6]]^S[6][x[8]]^S[7][x[9]]^S[6][x[3]];
93    roundKey[23]=S[4][x[5]]^S[5][x[4]]^S[6][x[10]]^S[7][x[11]]^S[7][x[7]];
94    //第 7 轮密钥计算
95    keyZ[0]=keyX[0]^S[4][x[13]]^S[5][x[15]]^S[6][x[12]]^S[7][x[14]]^S[6][x[8]];
96    word32ToByte(keyZ[0],z,0);
97    keyZ[1]=keyX[2]^S[4][z[0]]^S[5][z[2]]^S[6][z[1]]^S[7][z[3]]^S[7][x[10]];
98    word32ToByte(keyZ[1],z,1);
99    keyZ[2]=keyX[3]^S[4][z[7]]^S[5][z[6]]^S[6][z[5]]^S[7][z[4]]^S[4][x[9]];
100   word32ToByte(keyZ[2],z,2);
101   keyZ[3]=keyX[1]^S[4][z[10]]^S[5][z[9]]^S[6][z[11]]^S[7][z[8]]^S[5][x[11]];
102   word32ToByte(keyZ[3],z,3);
103   roundKey[24]=S[4][z[3]]^S[5][z[2]]^S[6][z[12]]^S[7][z[13]]^S[4][z[9]];
104   roundKey[25]=S[4][z[1]]^S[5][z[0]]^S[6][z[14]]^S[7][z[15]]^S[5][z[12]];
105   roundKey[26]=S[4][z[7]]^S[5][z[6]]^S[6][z[8]]^S[7][z[9]]^S[6][z[2]];
106   roundKey[27]=S[4][z[5]]^S[5][z[4]]^S[6][z[10]]^S[7][z[11]]^S[7][z[6]];
107   //第 8 轮密钥计算
108   keyX[0]=keyZ[2]^S[4][z[5]]^S[5][z[7]]^S[6][z[4]]^S[7][z[6]]^S[6][z[0]];
109   word32ToByte(keyX[0],x,0);
110   keyX[1]=keyZ[0]^S[4][x[0]]^S[5][x[2]]^S[6][x[1]]^S[7][x[3]]^S[7][z[2]];
111   word32ToByte(keyX[1],x,1);
112   keyX[2]=keyZ[1]^S[4][x[7]]^S[5][x[6]]^S[6][x[5]]^S[7][x[4]]^S[4][z[1]];
113   word32ToByte(keyX[2],x,2);
114   keyX[3]=keyZ[3]^S[4][x[10]]^S[5][x[9]]^S[6][x[11]]^S[7][x[8]]^S[5][z[3]];
115   word32ToByte(keyX[3],x,3);
116   roundKey[28]=S[4][x[8]]^S[5][x[9]]^S[6][x[7]]^S[7][x[6]]^S[4][x[3]];
117   roundKey[29]=S[4][x[10]]^S[5][x[11]]^S[6][x[5]]^S[7][x[4]]^S[5][x[7]];
118   roundKey[30]=S[4][x[12]]^S[5][x[13]]^S[6][x[3]]^S[7][x[2]]^S[6][x[8]];
119   roundKey[31]=S[4][x[14]]^S[5][x[15]]^S[6][x[1]]^S[7][x[0]]^S[7][x[13]];
120   for(i=0;i<16;i++)
121   {
122       keyM[i]=roundKey[i];
123       keyR[i]=roundKey[i+16]&0x1F;
124   }
125 }
```

函数的参数为输入的密钥，参数类型为 byte 型。

程序清单 8-2 中的第 4 行到第 7 行是将输入的密钥转换为 CAST_128 类中的成员变量，这部分内容也可以省去，在后续的密钥计算过程中也可以直接使用输入的密钥。

变量 keyX[4]是将输入的用于后续密钥计算中的临时变量，初始时是将输入的密钥以 word32 型数据存储输入的密钥，代码第 12 行到第 15 行为具体存储的过程，存储的方式如图 8-4 所示。

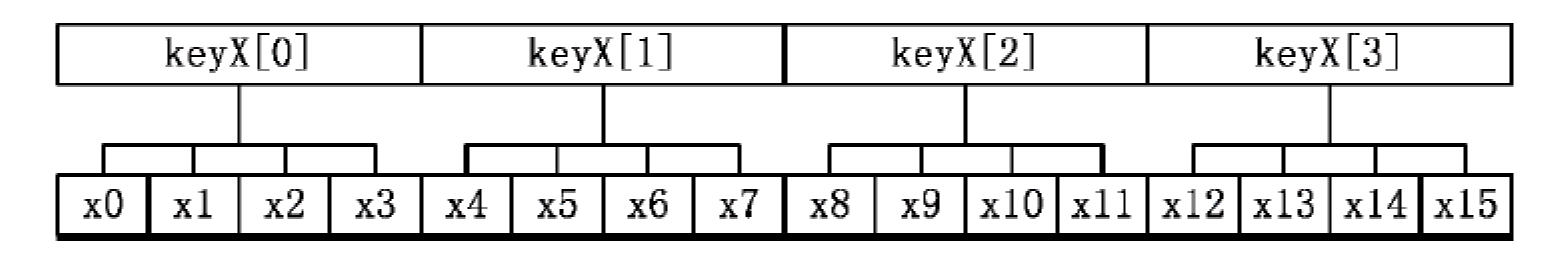


图 8-4 输入密钥-计算密钥数据转换示意图

例如 x0,x1,x2 和 x3 组成 keyX[0],组成方式为 x0 左移 24 位“或”x1 左移 16 位“或”x2 左移 8 位“或”x3,这样,就将 x1、x2、x3 和 x4 存放到 keyX[0]对应的位置,其他 keyX 处理的方法与 keyX[0]处理的方法相同。通过图 8-4 的方式,将输入的 16 个字节型密钥转换为 4 个 32 位 word32 型数据。

轮密钥的计算过程分为 8 轮,第 1 轮通过 keyX 计算 keyZ,然后再计算轮密钥,第 2 轮通过 keyZ 计算 keyX,然后再计算轮密钥。依次循环直至将 32 个轮密钥计算完毕。在计算过程中用到将 32 位的 word32 型数据分解为 8 位 byte 型数据,分解过程正好是图 8-4 所示过程的逆过程,该过程通过函数 word32ToByte(word32 in,byte bIn[],int position)来实现,该函数的详细代码见程序清单 8-3。

程序清单 8-3

```
void CAST_128::word32ToByte(word32 in,byte bIn[],int position)
{
    bIn[position*4]=(in>>24)&0xFF;
    bIn[position*4+1]=(in>>16)&0xFF;
    bIn[position*4+2]=(in>>8)&0xFF;
    bIn[position*4+3]=in&0xFF;
}
```

函数的输入为 word32 型数据的输入,以及数据处理的位置,输出为 4 个 byte 型数据。实现的方法是通过移位获得。

在密钥计算完成后通过程序清单 8-2 的第 120 行到 124 行代码将密钥转换为“掩码”密钥和“旋转”密钥。由于“旋转”密钥只需要使用最后 5 位,因此在运算过程中通过 &0x1F 的方法来获得最后 5 位数据,并将其他位的数据置为“0”。

在密钥的生成过程使用的 S 盒是第 5 个 S 盒到第 8 个 S 盒。

8.2.2 加密和解密

CAST_128 类的加密算法通过函数 encryption()来实现,函数 encryption()的具体代码见程序清单 8-4。

程序清单 8-4

```
word64 CAST_128::encryption(word64 in)
{
    word32 L[17],R[17],temp;
    L[0]=in>>32;
    R[0]=in&0xFFFFFFFF;
    int i;
```

```
07      for(i=1;i<=16;i++)
08      {
09          R[i]=L[i-1]^F(R[i-1],keyM[i-1],keyR[i-1],i);
10          L[i]=R[i-1];
11      }
12      return (word64(R[16])<<32)|L[16];
13  }
```

加密函数的参数是 word64 型数据，函数的返回类型也是 word64 型。函数是实现的基本过程是：将输入的 64 位数据分割成两部分，各为 32 位数据，数据分割通过代码行第 4 行和第 5 行代码完成，左半部分的数据直接将输入右移 32 位获得，右半部分的数据将输入与 0xFFFFFFFF 进行“&”运算获得，在分割完成后，通过代码行第 7 行到第 11 行进行 16 轮相同的加密，最后将得到的数据转化为 64 位数据合并输出。

CAST_128 类的解密算法通过函数 decryption()来实现，函数 decryption()的具体代码见程序清单 8-5。

程序清单 8-5

```
word64 CAST_128::decryption(word64 in)
{
    word32 L[17],R[17];
    L[16]=in>>32;
    R[16]=in&0xFFFFFFFF;
    int i;
    for(i=15;i>=0;i--)
    {
        R[i]=L[i+1]^F(R[i+1],keyM[i],keyR[i],i+1);
        L[i]=R[i+1];
    }
    return (word64(R[0])<<32)|L[0];
}
```

解密函数的参数和返回类型都是 word64 型数据。函数实现的基本过程与加密过程相似，但在密钥的使用顺序上正好与加密过程相反，在解密完成后将最后一轮的数据合并后输出。

在加密和解密过程中都使用了 F 函数，F 函数是加密和解密过程的核心。F 函数的具体实现过程见程序清单 8-6。

程序清单 8-6

```
word32 CAST_128::F(word32 in,word32 km,word32 kr,int round)
{
    word32 f;              //F函数的值
    word32 I;              //计算用临时变量
    byte Ia,Ib,Ic,Id;
    switch(round)
    {
```

```
            case 1:
            case 4:
            case 7:
            case 10:
            case 13:
            case 16:
                I=((km+in)<<kr)|((km+in)>>(32-kr));
                Ia=(I>>24)&0xFF;
                Ib=(I>>16)&0xFF;
                Ic=(I>>8)&0xFF;
                Id=I&0xFF;
                f=((S[0][Ia])^S[1][Ib])-S[2][Ic]+S[3][Id];
                break;
            case 2:
            case 5:
            case 8:
            case 11:
            case 14:
                I=((km^in)<<kr)|((km^in)>>(32-kr));
                Ia=(I>>24)&0xFF;
                Ib=(I>>16)&0xFF;
                Ic=(I>>8)&0xFF;
                Id=I&0xFF;
                f=((S[0][Ia]-S[1][Ib])+S[2][Ic])^S[3][Id];
                break;
            case 3:
            case 6:
            case 9:
            case 12:
            case 15:
                I=((km-in)<<kr)|((km-in)>>(32-kr));
                Ia=(I>>24)&0xFF;
                Ib=(I>>16)&0xFF;
                Ic=(I>>8)&0xFF;
                Id=I&0xFF;
                f=((S[0][Ia]+S[1][Ib])^S[2][Ic])-S[3][Id];
            default:
                break;
        }
        return f;
}
```

F 函数的参数为 word32 型的输入、当前轮次的“掩码”密钥、当前轮次的“旋转”密钥和当前加密或解密的轮次，*F* 函数的返回值类型也是 word32 型。

F 函数的计算方法与加密的轮次相关，不同的加密或解密轮次采用不同的计算方法，例

如第 14 行到第 20 行是针对第 1、4、7、10、13 和 16 轮时使用的计算方法，第 26 行到第 32 行是针对第 2、5、8、11 和 14 轮时使用的计算方法，以此类推。在各轮的计算过程中，首先通过输入的子密钥对和计算用数据（加密或解密数据的右 32 位）计算得到 I，然后将 I 按字节进行分解，获得 4 个字节，分别为 Ia、Ib、Ic 和 Id，这 4 个字节通过移位和“&”运算计算获得，最后通过 Ia、Ib、Ic 和 Id 确定 S 盒的数据，计算得到 f。

在 *F* 函数的运算过程中使用了循环移位，图 8-5 为 16 位的数据循环左移 6 位的示意图。

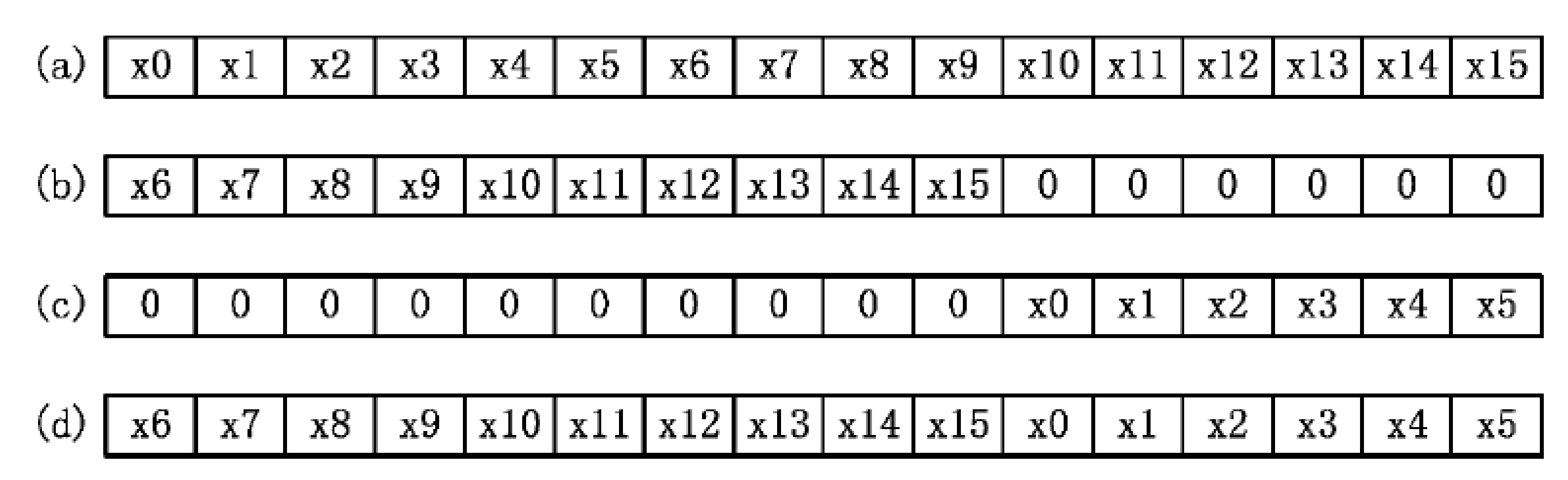


图 8-5 循环左移示意图

(a)为原始输入的数据，x0，x1，…，x15 为各位的数据，将输入 a 左移 6 位后得到(b)，将输入 a 右移(16－6)位后得到(c)，将(b)和(c)进行“或”运算得到(a)循环左移 6 位。

在 *F* 函数的运算过程中使用的 S 盒是第 1 个到第 4 个 S 盒。

8.2.3 数据初始化和程序测试

数据初始化主要是针对 S 盒进行初始化，程序测试可以通过主函数直接进行测试，主函数代码见程序清单 8-7。

程序清单 8-7

```
int main()
{
    CAST_128 cast_128;
    byte key[16]= {0x01,0x23,0x45,0x67,0x12,0x34,0x56,0x78,
                   0x23,0x45,0x67,0x89,0x34,0x56,0x78,0x9A};
    cast_128.getKey(key);
    word64 in= 0x0123456789abcdef;
    cout<< "The plainText= "<< hex<< in<< endl;
    word64 cipher= cast_128.encryption(in);
    cout<< "The cipherText= "<< hex<< cipher<< endl;
    word64 plain= cast_128.decryption(cipher);
    cout<< "The decipherText= "<< hex<< plain<< endl;
    return 0;
}
```

加密和解密测试结果如下。

```
The plainText= 123456789abcdef
The cipherText= 238b4fe5847e44b2
```

```
The decipherText=123456789abcdef
```

S 盒共有 8 组，每组 256 个数据。0～3 组用于 F 函数运算，4～7 组用于各轮子密钥运算。S 盒的初始化见程序清单 8-8。

程序清单 8-8

```
const word32 CAST_128::S[8][256]={
{0x30FB40D4,0x9FA0FF0B,0x6BECCD2F,0x3F258C7A,0x1E213F2F,0x9C004DD3,
0x6003E540,0xCF9FC949, 0xBFD4AF27,0x88BBBDB5,0xE2034090,0x98D09675,
0x6E63A0E0,0x15C361D2,0xC2E7661D,0x22D4FF8E,0x28683B6F,0xC07FD059,
0xFF2379C8,0x775F50E2,0x43C340D3,0xDF2F8656,0x887CA41A,0xA2D2BD2D,
0xA1C9E0D6,0x346C4819,0x61B76D87,0x22540F2F,0x2ABE32E1,0xAA54166B,
0x22568E3A,0xA2D341D0,0x66DB40C8,0xA784392F,0x004DFF2F,0x2DB9D2DE,
0x97943FAC,0x4A97C1D8,0x527644B7,0xB5F437A7,0xB82CBAEF,0xD751D159,
0x6FF7F0ED,0x5A097A1F,0x827B68D0,0x90ECF52E,0x22B0C054,0xBC8E5935,
0x4B6D2F7F,0x50BB64A2,0xD2664910,0xBEE5812D,0xB7332290,0xE93B159F,
0xB48EE411,0x4BFF345D,0xFD45C240,0xAD31973F,0xC4F6D02E,0x55FC8165,
0xD5B1CAAD,0xA1AC2DAE,0xA2D4B76D,0xC19B0C50,0x882240F2,0x0C6E4F38,
0xA4E4BFD7,0x4F5BA272,0x564C1D2F,0xC59C5319,0xB949E354,0xB04669FE,
0xB1B6AB8A,0xC71358DD,0x6385C545,0x110F935D,0x57538AD5,0x6A390493,
0xE63D37E0,0x2A54F6B3,0x3A787D5F,0x6276A0B5,0x19A6FCDF,0x7A42206A,
0x29F9D4D5,0xF61B1891,0xBB72275E,0xAA508167,0x38901091,0xC6B505EB,
0x84C7CB8C,0x2AD75A0F,0x874A1427,0xA2D1936B,0x2AD286AF,0xAA56D291,
0xD7894360,0x425C750D,0x93B39E26,0x187184C9,0x6C00B32D,0x73E2BB14,
0xA0BEBC3C,0x54623779,0x64459EAB,0x3F328B82,0x7718CF82,0x59A2CEA6,
0x04EE002E,0x89FE78E6,0x3FAB0950,0x325FF6C2,0x81383F05,0x6963C5C8,
0x76CB5AD6,0xD49974C9,0xCA180DCF,0x380782D5,0xC7FA5CF6,0x8AC31511,
0x35E79E13,0x47DA91D0,0xF40F9086,0xA7E2419E,0x31366241,0x051EF495,
0xAA573B04,0x4A805D8D,0x548300D0,0x00322A3C,0xBF64CDDF,0xBA57A68E,
0x75C6372B,0x50AFD341,0xA7C13275,0x915A0BF5,0x6B54BFAB,0x2B0B1426,
0xAB4CC9D7,0x449CCD82,0xF7FBF265,0xAB85C5F3,0x1B55DB94,0xAAD4E324,
0xCFA4BD3F,0x2DEAA3E2,0x9E204D02,0xC8BD25AC,0xEADF55B3,0xD5BD9E98,
0xE31231B2,0x2AD5AD6C,0x954329DE,0xADBE4528,0xD8710F69,0xAA51C90F,
0xAA786BF6,0x22513F1E,0xAA51A79B,0x2AD344CC,0x7B5A41F0,0xD37CFBAD,
0x1B069505,0x41ECE491,0xB4C332E6,0x032268D4,0xC9600ACC,0xCE387E6D,
0xBF6BB16C,0x6A70FB78,0x0D03D9C9,0xD4DF39DE,0xE01063DA,0x4736F464,
0x5AD328D8,0xB347CC96,0x75BB0FC3,0x98511BFB,0x4FFBCC35,0xB58BCF6A,
0xE11F0ABC,0xBFC5FE4A,0xA70AEC10,0xAC39570A,0x3F04442F,0x6188B153,
0xE0397A2E,0x5727CB79,0x9CEB418F,0x1CACD68D,0x2AD37C96,0x0175CB9D,
0xC69DFF09,0xC75B65F0,0xD9DB40D8,0xEC0E7779,0x4744EAD4,0xB11C3274,
0xDD24CB9E,0x7E1C54BD,0xF01144F9,0xD2240EB1,0x9675B3FD,0xA3AC3755,
0xD47C27AF,0x51C85F4D,0x56907596,0xA5BB15E6,0x580304F0,0xCA042CF1,
0x011A37EA,0x8DBFAADB,0x35BA3E4A,0x3526FFA0,0xC37B4D09,0xBC306ED9,
0x98A52666,0x5648F725,0xFF5E569D,0x0CED63D0,0x7C63B2CF,0x700B45E1,
0xD5EA50F1,0x85A92872,0xAF1FBDA7,0xD4234870,0xA7870BF3,0x2D3B4D79,
0x42E04198,0x0CD0EDE7,0x26470DB8,0xF881814C,0x474D6AD7,0x7C0C5E5C,
```

```
0xD1231959,0x381B7298,0xF5D2F4DB,0xAB838653,0x6E2F1E23,0x83719C9E,
0xBD91E046,0x9A56456E,0xDC39200C,0x20C8C571,0x962BDA1C,0xE1E696FF,
0xB141AB08,0x7CCA89B9,0x1A69E783,0x02CC4843,0xA2F7C579,0x429EF47D,
0x427B169C,0x5AC9F049,0xDD8F0F00,0x5C8165BF},          //S[0]

{0x1F201094,0xEF0BA75B,0x69E3CF7E,0x393F4380,0xFE61CF7A,0xEEC5207A,
0x55889C94,0x72FC0651, 0xADA7EF79,0x4E1D7235,0xD55A63CE,0xDE0436BA,
0x99C430EF,0x5F0C0794,0x18DCDB7D,0xA1D6EFF3,0xA0B52F7B,0x59E83605,
0xEE15B094,0xE9FFD909,0xDC440086,0xEF944459,0xBA83CCB3,0xE0C3CDFB,
0xD1DA4181,0x3B092AB1,0xF997F1C1,0xA5E6CF7B,0x01420DDB,0xE4E7EF5B,
0x25A1FF41,0xE180F806,0x1FC41080,0x179BEE7A,0xD37AC6A9,0xFE5830A4,
0x98DE8B7F,0x77E83F4E,0x79929269,0x24FA9F7B,0xE113C85B,0xACC40083,
0xD7503525,0xF7EA615F,0x62143154,0x0D554B63,0x5D681121,0xC866C359,
0x3D63CF73,0xCEE234C0,0xD4D87E87,0x5C672B21,0x071F6181,0x39F7627F,
0x361E3084,0xE4EB573B,0x602F64A4,0xD63ACD9C,0x1BBC4635,0x9E81032D,
0x2701F50C,0x99847AB4,0xA0E3DF79,0xBA6CF38C,0x10843094,0x2537A95E,
0xF46F6FFE,0xA1FF3B1F,0x208CFB6A,0x8F458C74,0xD9E0A227,0x4EC73A34,
0xFC884F69,0x3E4DE8DF,0xEF0E0088,0x3559648D,0x8A45388C,0x1D804366,
0x721D9BFD,0xA58684BB,0xE8256333,0x844E8212,0x128D8098,0xFED33FB4,
0xCE280AE1,0x27E19BA5,0xD5A6C252,0xE49754BD,0xC5D655DD,0xEB667064,
0x77840B4D,0xA1B6A801,0x84DB26A9,0xE0B56714,0x21F043B7,0xE5D05860,
0x54F03084,0x066FF472,0xA31AA153,0xDADC4755,0xB5625DBF,0x68561BE6,
0x83CA6B94,0x2D6ED23B,0xECCF01DB,0xA6D3D0BA,0xB6803D5C,0xAF77A709,
0x33B4A34C,0x397BC8D6,0x5EE22B95,0x5F0E5304,0x81ED6F61,0x20E74364,
0xB45E1378,0xDE18639B,0x881CA122,0xB96726D1,0x8049A7E8,0x22B7DA7B,
0x5E552D25,0x5272D237,0x79D2951C,0xC60D894C,0x488CB402,0x1BA4FE5B,
0xA4B09F6B,0x1CA815CF,0xA20C3005,0x8871DF63,0xB9DE2FCB,0x0CC6C9E9,
0x0BEEFF53,0xE3214517,0xB4542835,0x9F63293C,0xEE41E729,0x6E1D2D7C,
0x50045286,0x1E6685F3,0xF33401C6,0x30A22C95,0x31A70850,0x60930F13,
0x73F98417,0xA1269859,0xEC645C44,0x52C877A9,0xCDFF33A6,0xA02B1741,
0x7CBAD9A2,0x2180036F,0x50D99C08,0xCB3F4861,0xC26BD765,0x64A3F6AB,
0x80342676,0x25A75E7B,0xE4E6D1FC,0x20C710E6,0xCDF0B680,0x17844D3B,
0x31EEF84D,0x7E0824E4,0x2CCB49EB,0x846A3BAE,0x8FF77888,0xEE5D60F6,
0x7AF75673,0x2FDD5CDB,0xA11631C1,0x30F66F43,0xB3FAEC54,0x157FD7FA,
0xEF8579CC,0xD152DE58,0xDB2FFD5E,0x8F32CE19,0x306AF97A,0x02F03EF8,
0x99319AD5,0xC242FA0F,0xA7E3EBB0,0xC68E4906,0xB8DA230C,0x80823028,
0xDCDEF3C8,0xD35FB171,0x088A1BC8,0xBEC0C560,0x61A3C9E8,0xBCA8F54D,
0xC72FEFFA,0x22822E99,0x82C570B4,0xD8D94E89,0x8B1C34BC,0x301E16E6,
0x273BE979,0xB0FFEAA6,0x61D9B8C6,0x00B24869,0xB7FFCE3F,0x08DC283B,
0x43DAF65A,0xF7E19798,0x7619B72F,0x8F1C9BA4,0xDC8637A0,0x16A7D3B1,
0x9FC393B7,0xA7136EEB,0xC6BCC63E,0x1A513742,0xEF6828BC,0x520365D6,
0x2D6A77AB,0x3527ED4B,0x821FD216,0x095C6E2E,0xDB92F2FB,0x5EEA29CB,
0x145892F5,0x91584F7F,0x5483697B,0x2667A8CC,0x85196048,0x8C4BACEA,
0x833860D4,0x0D23E0F9,0x6C387E8A,0x0AE6D249,0xB284600C,0xD835731D,
0xDCB1C647,0xAC4C56EA,0x3EBD81B3,0x230EABB0,0x6438BC87,0xF0B5B1FA,
```

```
0x8F5EA2B3,0xFC184642,0x0A036B7A,0x4FB089BD,0x649DA589,0xA345415E,
0x5C038323,0x3E5D3BB9,0x43D79572,0x7E6DD07C,0x06DFDF1E,0x6C6CC4EF,
0x7160A539,0x73BFBE70,0x83877605,0x4523ECF1},           //S[1]

{0x8DEFC240,0x25FA5D9F,0xEB903DBF,0xE810C907,0x47607FFF,0x369FE44B,
0x8C1FC644,0xAECECA90,0xBEB1F9BF,0xEEFBCAEA,0xE8CF1950,0x51DF07AE,
0x920E8806,0xF0AD0548,0xE13C8D83,0x927010D5,0x11107D9F,0x07647DB9,
0xB2E3E4D4,0x3D4F285E,0xB9AFA820,0xFADE82E0,0xA067268B,0x8272792E,
0x553FB2C0,0x489AE22B,0xD4EF9794,0x125E3FBC,0x21FFFCEE,0x825B1BFD,
0x9255C5ED,0x1257A240,0x4E1A8302,0xBAE07FFF,0x528246E7,0x8E57140E,
0x3373F7BF,0x8C9F8188,0xA6FC4EE8,0xC982B5A5,0xA8C01DB7,0x579FC264,
0x67094F31,0xF2BD3F5F,0x40FFF7C1,0x1FB78DFC,0x8E6BD2C1,0x437BE59B,
0x99B03DBF,0xB5DBC64B,0x638DC0E6,0x55819D99,0xA197C81C,0x4A012D6E,
0xC5884A28,0xCCC36F71,0xB843C213,0x6C0743F1,0x8309893C,0x0FEDDD5F,
0x2F7FE850,0xD7C07F7E,0x02507FBF,0x5AFB9A04,0xA747D2D0,0x1651192E,
0xAF70BF3E,0x58C31380,0x5F98302E,0x727CC3C4,0x0A0FB402,0x0F7FEF82,
0x8C96FDAD,0x5D2C2AAE,0x8EE99A49,0x50DA88B8,0x8427F4A0,0x1EAC5790,
0x796FB449,0x8252DC15,0xEFBD7D9B,0xA672597D,0xADA840D8,0x45F54504,
0xFA5D7403,0xE83EC305,0x4F91751A,0x925669C2,0x23EFE941,0xA903F12E,
0x60270DF2,0x0276E4B6,0x94FD6574,0x927985B2,0x8276DBCB,0x02778176,
0xF8AF918D,0x4E48F79E,0x8F616DDF,0xE29D840E,0x842F7D83,0x340CE5C8,
0x96BBB682,0x93B4B148,0xEF303CAB,0x984FAF28,0x779FAF9B,0x92DC560D,
0x224D1E20,0x8437AA88,0x7D29DC96,0x2756D3DC,0x8B907CEE,0xB51FD240,
0xE7C07CE3,0xE566B4A1,0xC3E9615E,0x3CF8209D,0x6094D1E3,0xCD9CA341,
0x5C76460E,0x00EA983B,0xD4D67881,0xFD47572C,0xF76CEDD9,0xBDA8229C,
0x127DADAA,0x438A074E,0x1F97C090,0x081BDB8A,0x93A07EBE,0xB938CA15,
0x97B03CFF,0x3DC2C0F8,0x8D1AB2EC,0x64380E51,0x68CC7BFB,0xD90F2788,
0x12490181,0x5DE5FFD4,0xDD7EF86A,0x76A2E214,0xB9A40368,0x925D958F,
0x4B39FFFA,0xBA39AEE9,0xA4FFD30B,0xFAF7933B,0x6D498623,0x193CBCFA,
0x27627545,0x825CF47A,0x61BD8BA0,0xD11E42D1,0xCEAD04F4,0x127EA392,
0x10428DB7,0x8272A972,0x9270C4A8,0x127DE50B,0x285BA1C8,0x3C62F44F,
0x35C0EAA5,0xE805D231,0x428929FB,0xB4FCDF82,0x4FB66A53,0x0E7DC15B,
0x1F081FAB,0x108618AE,0xFCFD086D,0xF9FF2889,0x694BCC11,0x236A5CAE,
0x12DECA4D,0x2C3F8CC5,0xD2D02DFE,0xF8EF5896,0xE4CF52DA,0x95155B67,
0x494A488C,0xB9B6A80C,0x5C8F82BC,0x89D36B45,0x3A609437,0xEC00C9A9,
0x44715253,0x0A874B49,0xD773BC40,0x7C34671C,0x02717EF6,0x4FEB5536,
0xA2D02FFF,0xD2BF60C4,0xD43F03C0,0x50B4EF6D,0x07478CD1,0x006E1888,
0xA2E53F55,0xB9E6D4BC,0xA2048016,0x97573833,0xD7207D67,0xDE0F8F3D,
0x72F87B33,0xABCC4F33,0x7688C55D,0x7B00A6B0,0x947B0001,0x570075D2,
0xF9BB88F8,0x8942019E,0x4264A5FF,0x856302E0,0x72DBD92B,0xEE971B69,
0x6EA22FDE,0x5F08AE2B,0xAF7A616D,0xE5C98767,0xCF1FEBD2,0x61EFC8C2,
0xF1AC2571,0xCC8239C2,0x67214CB8,0xB1E583D1,0xB7DC3E62,0x7F10BDCE,
0xF90A5C38,0x0FF0443D,0x606E6DC6,0x60543A49,0x5727C148,0x2BE98A1D,
0x8AB41738,0x20E1BE24,0xAF96DA0F,0x68458425,0x99833BE5,0x600D457D,
0x282F9350,0x8334B362,0xD91D1120,0x2B6D8DA0,0x642B1E31,0x9C305A00,
```

```
0x52BCE688,0x1B03588A,0xF7BAEFD5,0x4142ED9C,0xA4315C11,0x83323EC5,
0xDFEF4636,0xA133C501,0xE9D3531C,0xEE353783},          //S[2]

{0x9DB30420,0x1FB6E9DE,0xA7BE7BEF,0xD273A298,0x4A4F7BDB,0x64AD8C57,
0x85510443,0xFA020ED1,0x7E287AFF,0xE60FB663,0x095F35A1,0x79EBF120,
0xFD059D43,0x6497B7B1,0xF3641F63,0x241E4ADF,0x28147F5F,0x4FA2B8CD,
0xC9430040,0x0CC32220,0xFDD30B30,0xC0A5374F,0x1D2D00D9,0x24147B15,
0xEE4D111A,0x0FCA5167,0x71FF904C,0x2D195FFE,0x1A05645F,0x0C13FEFE,
0x081B08CA,0x05170121,0x80530100,0xE83E5EFE,0xAC9AF4F8,0x7FE72701,
0xD2B8EE5F,0x06DF4261,0xBB9E9B8A,0x7293EA25,0xCE84FFDF,0xF5718801,
0x3DD64B04,0xA26F263B,0x7ED48400,0x547EEBE6,0x446D4CA0,0x6CF3D6F5,
0x2649ABDF,0xAEA0C7F5,0x36338CC1,0x503F7E93,0xD3772061,0x11B638E1,
0x72500E03,0xF80EB2BB,0xABE0502E,0xEC8D77DE,0x57971E81,0xE14F6746,
0xC9335400,0x6920318F,0x081DBB99,0xFFC304A5,0x4D351805,0x7F3D5CE3,
0xA6C866C6,0x5D5BCCA9,0xDAEC6FEA,0x9F926F91,0x9F46222F,0x3991467D,
0xA5BF6D8E,0x1143C44F,0x43958302,0xD0214EEB,0x022083B8,0x3FB6180C,
0x18F8931E,0x281658E6,0x26486E3E,0x8BD78A70,0x7477E4C1,0xB506E07C,
0xF32D0A25,0x79098B02,0xE4EABB81,0x28123B23,0x69DEAD38,0x1574CA16,
0xDF871B62,0x211C40B7,0xA51A9EF9,0x0014377B,0x041E8AC8,0x09114003,
0xBD59E4D2,0xE3D156D5,0x4FE876D5,0x2F91A340,0x557BE8DE,0x00EAE4A7,
0x0CE5C2EC,0x4DB4BBA6,0xE756BDFF,0xDD3369AC,0xEC17B035,0x06572327,
0x99AFC8B0,0x56C8C391,0x6B65811C,0x5E146119,0x6E85CB75,0xBE07C002,
0xC2325577,0x893FF4EC,0x5BBFC92D,0xD0EC3B25,0xB7801AB7,0x8D6D3B24,
0x20C763EF,0xC366A5FC,0x9C382880,0x0ACE3205,0xAAC9548A,0xECA1D7C7,
0x041AFA32,0x1D16625A,0x6701902C,0x9B757A54,0x31D477F7,0x9126B031,
0x36CC6FDB,0xC70B8B46,0xD9E66A48,0x56E55A79,0x026A4CEB,0x52437EFF,
0x2F8F76B4,0x0DF980A5,0x8674CDE3,0xEDDA04EB,0x17A9BE04,0x2C18F4DF,
0xB7747F9D,0xAB2AF7B4,0xEFC34D20,0x2E096B7C,0x1741A254,0xE5B6A035,
0x213D42F6,0x2C1C7C26,0x61C2F50F,0x6552DAF9,0xD2C231F8,0x25130F69,
0xD8167FA2,0x0418F2C8,0x001A96A6,0x0D1526AB,0x63315C21,0x5E0A72EC,
0x49BAFEFD,0x187908D9,0x8D0DBD86,0x311170A7,0x3E9B640C,0xCC3E10D7,
0xD5CAD3B6,0x0CAEC388,0xF73001E1,0x6C728AFF,0x71EAE2A1,0x1F9AF36E,
0xCFCBD12F,0xC1DE8417,0xAC07BE6B,0xCB44A1D8,0x8B9B0F56,0x013988C3,
0xB1C52FCA,0xB4BE31CD,0xD8782806,0x12A3A4E2,0x6F7DE532,0x58FD7EB6,
0xD01EE900,0x24ADFFC2,0xF4990FC5,0x9711AAC5,0x001D7B95,0x82E5E7D2,
0x109873F6,0x00613096,0xC32D9521,0xADA121FF,0x29908415,0x7FBB977F,
0xAF9EB3DB,0x29C9ED2A,0x5CE2A465,0xA730F32C,0xD0AA3FE8,0x8A5CC091,
0xD49E2CE7,0x0CE454A9,0xD60ACD86,0x015F1919,0x77079103,0xDEA03AF6,
0x78A8565E,0xDEE356DF,0x21F05CBE,0x8B75E387,0xB3C50651,0xB8A5C3EF,
0xD8EEB6D2,0xE523BE77,0xC2154529,0x2F69EFDF,0xAFE67AFB,0xF470C4B2,
0xF3E0EB5B,0xD6CC9876,0x39E4460C,0x1FDA8538,0x1987832F,0xCA007367,
0xA99144F8,0x296B299E,0x492FC295,0x9266BEAB,0xB5676E69,0x9BD3DDDA,
0xDF7E052F,0xDB25701C,0x1B5E51EE,0xF65324E6,0x6AFCE36C,0x0316CC04,
0x8644213E,0xB7DC59D0,0x7965291F,0xCCD6FD43,0x41823979,0x932BCDF6,
0xB657C34D,0x4EDFD282,0x7AE5290C,0x3CB9536B,0x851E20FE,0x9833557E,
```

```
0x13ECF0B0,0xD3FFB372,0x3F85C5C1,0x0AEF7ED2},          //S[3]

{0x7EC90C04,0x2C6E74B9,0x9B0E66DF,0xA6337911,0xB86A7FFF,0x1DD358F5,
0x44DD9D44,0x1731167F,0x08FBF1FA,0xE7F511CC,0xD2051B00,0x735ABA00,
0x2AB722D8,0x386381CB,0xACF6243A,0x69BEFD7A,0xE6A2E77F,0xF0C720CD,
0xC4494816,0xCCF5C180,0x38851640,0x15B0A848,0xE68B18CB,0x4CAADEFF,
0x5F480A01,0x0412B2AA,0x259814FC,0x41D0EFE2,0x4E40B48D,0x248EB6FB,
0x8DBA1CFE,0x41A99B02,0x1A550A04,0xBA8F65CB,0x7251F4E7,0x95A51725,
0xC106ECD7,0x97A5980A,0xC539B9AA,0x4D79FE6A,0xF2F3F763,0x68AF8040,
0xED0C9E56,0x11B4958B,0xE1EB5A88,0x8709E6B0,0xD7E07156,0x4E29FEA7,
0x6366E52D,0x02D1C000,0xC4AC8E05,0x9377F571,0x0C05372A,0x578535F2,
0x2261BE02,0xD642A0C9,0xDF13A280,0x74B55BD2,0x682199C0,0xD421E5EC,
0x53FB3CE8,0xC8ADEDB3,0x28A87FC9,0x3D959981,0x5C1FF900,0xFE38D399,
0x0C4EFF0B,0x062407EA,0xAA2F4FB1,0x4FB96976,0x90C79505,0xB0A8A774,
0xEF55A1FF,0xE59CA2C2,0xA6B62D27,0xE66A4263,0xDF65001F,0x0EC50966,
0xDFDD55BC,0x29DE0655,0x911E739A,0x17AF8975,0x32C7911C,0x89F89468,
0x0D01E980,0x524755F4,0x03B63CC9,0x0CC844B2,0xBCF3F0AA,0x87AC36E9,
0xE53A7426,0x01B3D82B,0x1A9E7449,0x64EE2D7E,0xCDDBB1DA,0x01C94910,
0xB868BF80,0x0D26F3FD,0x9342EDE7,0x04A5C284,0x636737B6,0x50F5B616,
0xF24766E3,0x8ECA36C1,0x136E05DB,0xFEF18391,0xFB887A37,0xD6E7F7D4,
0xC7FB7DC9,0x3063FCDF,0xB6F589DE,0xEC2941DA,0x26E46695,0xB7566419,
0xF654EFC5,0xD08D58B7,0x48925401,0xC1BACB7F,0xE5FF550F,0xB6083049,
0x5BB5D0E8,0x87D72E5A,0xAB6A6EE1,0x223A66CE,0xC62BF3CD,0x9E0885F9,
0x68CB3E47,0x086C010F,0xA21DE820,0xD18B69DE,0xF3F65777,0xFA02C3F6,
0x407EDAC3,0xCBB3D550,0x1793084D,0xB0D70EBA,0x0AB378D5,0xD951FB0C,
0xDED7DA56,0x4124BBE4,0x94CA0B56,0x0F5755D1,0xE0E1E56E,0x6184B5BE,
0x580A249F,0x94F74BC0,0xE327888E,0x9F7B5561,0xC3DC0280,0x05687715,
0x646C6BD7,0x44904DB3,0x66B4F0A3,0xC0F1648A,0x697ED5AF,0x49E92FF6,
0x309E374F,0x2CB6356A,0x85808573,0x4991F840,0x76F0AE02,0x083BE84D,
0x28421C9A,0x44489406,0x736E4CB8,0xC1092910,0x8BC95FC6,0x7D869CF4,
0x134F616F,0x2E77118D,0xB31B2BE1,0xAA90B472,0x3CA5D717,0x7D161BBA,
0x9CAD9010,0xAF462BA2,0x9FE459D2,0x45D34559,0xD9F2DA13,0xDBC65487,
0xF3E4F94E,0x176D486F,0x097C13EA,0x631DA5C7,0x445F7382,0x175683F4,
0xCDC66A97,0x70BE0288,0xB3CDCF72,0x6E5DD2F3,0x20936079,0x459B80A5,
0xBE60E2DB,0xA9C23101,0xEBA5315C,0x224E42F2,0x1C5C1572,0xF6721B2C,
0x1AD2FFF3,0x8C25404E,0x324ED72F,0x4067B7FD,0x0523138E,0x5CA3BC78,
0xDC0FD66E,0x75922283,0x784D6B17,0x58EBB16E,0x44094F85,0x3F481D87,
0xFCFEAE7B,0x77B5FF76,0x8C2302BF,0xAAF47556,0x5F46B02A,0x2B092801,
0x3D38F5F7,0x0CA81F36,0x52AF4A8A,0x66D5E7C0,0xDF3B0874,0x95055110,
0x1B5AD7A8,0xF61ED5AD,0x6CF6E479,0x20758184,0xD0CEFA65,0x88F7BE58,
0x4A046826,0x0FF6F8F3,0xA09C7F70,0x5346ABA0,0x5CE96C28,0xE176EDA3,
0x6BAC307F,0x376829D2,0x85360FA9,0x17E3FE2A,0x24B79767,0xF5A96B20,
0xD6CD2595,0x68FF1EBF,0x7555442C,0xF19F06BE,0xF9E0659A,0xEEB9491D,
0x34010718,0xBB30CAB8,0xE822FE15,0x88570983,0x750E6249,0xDA627E55,
0x5E76FFA8,0xB1534546,0x6D47DE08,0xEFE9E7D4},          //S[4]
```

```

{0xF6FA8F9D,0x2CAC6CE1,0x4CA34867,0xE2337F7C,0x95DB08E7,0x016843B4,
0xECED5CBC,0x325553AC,0xBF9F0960,0xDFA1E2ED,0x83F0579D,0x63ED86B9,
0x1AB6A6B8,0xDE5EBE39,0xF38FF732,0x8989B138,0x33F14961,0xC01937BD,
0xF506C6DA,0xE4625E7E,0xA308EA99,0x4E23E33C,0x79CBD7CC,0x48A14367,
0xA3149619,0xFEC94BD5,0xA114174A,0xEAA01866,0xA084DB2D,0x09A8486F,
0xA888614A,0x2900AF98,0x01665991,0xE1992863,0xC8F30C60,0x2E78EF3C,
0xD0D51932,0xCF0FEC14,0xF7CA07D2,0xD0A82072,0xFD41197E,0x9305A6B0,
0xE86BE3DA,0x74BED3CD,0x372DA53C,0x4C7F4448,0xDAB5D440,0x6DBA0EC3,
0x083919A7,0x9FBAEED9,0x49DBCFB0,0x4E670C53,0x5C3D9C01,0x64BDB941,
0x2C0E636A,0xBA7DD9CD,0xEA6F7388,0xE70BC762,0x35F29ADB,0x5C4CDD8D,
0xF0D48D8C,0xB88153E2,0x08A19866,0x1AE2EAC8,0x284CAF89,0xAA928223,
0x9334BE53,0x3B3A21BF,0x16434BE3,0x9AEA3906,0xEFE8C36E,0xF890CDD9,
0x80226DAE,0xC340A4A3,0xDF7E9C09,0xA694A807,0x5B7C5ECC,0x221DB3A6,
0x9A69A02F,0x68818A54,0xCEB2296F,0x53C0843A,0xFE893655,0x25BFE68A,
0xB4628ABC,0xCF222EBF,0x25AC6F48,0xA9A99387,0x53BDDB65,0xE76FFBE7,
0xE967FD78,0x0BA93563,0x8E342BC1,0xE8A11BE9,0x4980740D,0xC8087DFC,
0x8DE4BF99,0xA11101A0,0x7FD37975,0xDA5A26C0,0xE81F994F,0x9528CD89,
0xFD339FED,0xB87834BF,0x5F04456D,0x22258698,0xC9C4C83B,0x2DC156BE,
0x4F628DAA,0x57F55EC5,0xE2220ABE,0xD2916EBF,0x4EC75B95,0x24F2C3C0,
0x42D15D99,0xCD0D7FA0,0x7B6E27FF,0xA8DC8AF0,0x7345C106,0xF41E232F,
0x35162386,0xE6EA8926,0x3333B094,0x157EC6F2,0x372B74AF,0x692573E4,
0xE9A9D848,0xF3160289,0x3A62EF1D,0xA787E238,0xF3A5F676,0x74364853,
0x20951063,0x4576698D,0xB6FAD407,0x592AF950,0x36F73523,0x4CFB6E87,
0x7DA4CEC0,0x6C152DAA,0xCB0396A8,0xC50DFE5D,0xFCD707AB,0x0921C42F,
0x89DFF0BB,0x5FE2BE78,0x448F4F33,0x754613C9,0x2B05D08D,0x48B9D585,
0xDC049441,0xC8098F9B,0x7DEDE786,0xC39A3373,0x42410005,0x6A091751,
0x0EF3C8A6,0x890072D6,0x28207682,0xA9A9F7BE,0xBF32679D,0xD45B5B75,
0xB353FD00,0xCBB0E358,0x830F220A,0x1F8FB214,0xD372CF08,0xCC3C4A13,
0x8CF63166,0x061C87BE,0x88C98F88,0x6062E397,0x47CF8E7A,0xB6C85283,
0x3CC2ACFB,0x3FC06976,0x4E8F0252,0x64D8314D,0xDA3870E3,0x1E665459,
0xC10908F0,0x513021A5,0x6C5B68B7,0x822F8AA0,0x3007CD3E,0x74719EEF,
0xDC872681,0x073340D4,0x7E432FD9,0x0C5EC241,0x8809286C,0xF592D891,
0x08A930F6,0x957EF305,0xB7FBFFBD,0xC266E96F,0x6FE4AC98,0xB173ECC0,
0xBC60B42A,0x953498DA,0xFBA1AE12,0x2D4BD736,0x0F25FAAB,0xA4F3FCEB,
0xE2969123,0x257F0C3D,0x9348AF49,0x361400BC,0xE8816F4A,0x3814F200,
0xA3F94043,0x9C7A54C2,0xBC704F57,0xDA41E7F9,0xC25AD33A,0x54F4A084,
0xB17F5505,0x59357CBE,0xEDBD15C8,0x7F97C5AB,0xBA5AC7B5,0xB6F6DEAF,
0x3A479C3A,0x5302DA25,0x653D7E6A,0x54268D49,0x51A477EA,0x5017D55B,
0xD7D25D88,0x44136C76,0x0404A8C8,0xB8E5A121,0xB81A928A,0x60ED5869,
0x97C55B96,0xEAEC991B,0x29935913,0x01FDB7F1,0x088E8DFA,0x9AB6F6F5,
0x3B4CBF9F,0x4A5DE3AB,0xE6051D35,0xA0E1D855,0xD36B4CF1,0xF544EDEB,
0xB0E93524,0xBEBB8FBD,0xA2D762CF,0x49C92F54,0x38B5F331,0x7128A454,
0x48392905,0xA65B1DB8,0x851C97BD,0xD675CF2F},          //S[5]

```

```
{0x85E04019,0x332BF567,0x662DBFFF,0xCFC65693,0x2A8D7F6F,0xAB9BC912,
0xDE6008A1,0x2028DA1F,0x0227BCE7,0x4D642916,0x18FAC300,0x50F18B82,
0x2CB2CB11,0xB232E75C,0x4B3695F2,0xB28707DE,0xA05FBCF6,0xCD4181E9,
0xE150210C,0xE24EF1BD,0xB168C381,0xFDE4E789,0x5C79B0D8,0x1E8BFD43,
0x4D495001,0x38BE4341,0x913CEE1D,0x92A79C3F,0x089766BE,0xBAEEADF4,
0x1286BECF,0xB6EACB19,0x2660C200,0x7565BDE4,0x64241F7A,0x8248DCA9,
0xC3B3AD66,0x28136086,0x0BD8DFA8,0x356D1CF2,0x107789BE,0xB3B2E9CE,
0x0502AA8F,0x0BC0351E,0x166BF52A,0xEB12FF82,0xE3486911,0xD34D7516,
0x4E7B3AFF,0x5F43671B,0x9CF6E037,0x4981AC83,0x334266CE,0x8C9341B7,
0xD0D854C0,0xCB3A6C88,0x47BC2829,0x4725BA37,0xA66AD22B,0x7AD61F1E,
0x0C5CBAFA,0x4437F107,0xB6E79962,0x42D2D816,0x0A961288,0xE1A5C06E,
0x13749E67,0x72FC081A,0xB1D139F7,0xF9583745,0xCF19DF58,0xBEC3F756,
0xC06EBA30,0x07211B24,0x45C28829,0xC95E317F,0xBC8EC511,0x38BC46E9,
0xC6E6FA14,0xBAE8584A,0xAD4EBC46,0x468F508B,0x7829435F,0xF124183B,
0x821DBA9F,0xAFF60FF4,0xEA2C4E6D,0x16E39264,0x92544A8B,0x009B4FC3,
0xABA68CED,0x9AC96F78,0x06A5B79A,0xB2856E6E,0x1AEC3CA9,0xBE838688,
0x0E0804E9,0x55F1BE56,0xE7E5363B,0xB3A1F25D,0xF7DEBB85,0x61FE033C,
0x16746233,0x3C034C28,0xDA6D0C74,0x79AAC56C,0x3CE4E1AD,0x51F0C802,
0x98F8F35A,0x1626A49F,0xEED82B29,0x1D382FE3,0x0C4FB99A,0xBB325778,
0x3EC6D97B,0x6E77A6A9,0xCB658B5C,0xD45230C7,0x2BD1408B,0x60C03EB7,
0xB9068D78,0xA33754F4,0xF430C87D,0xC8A71302,0xB96D8C32,0xEBD4E7BE,
0xBE8B9D2D,0x7979FB06,0xE7225308,0x8B75CF77,0x11EF8DA4,0xE083C858,
0x8D6B786F,0x5A6317A6,0xFA5CF7A0,0x5DDA0033,0xF28EBFB0,0xF5B9C310,
0xA0EAC280,0x08B9767A,0xA3D9D2B0,0x79D34217,0x021A718D,0x9AC6336A,
0x2711FD60,0x438050E3,0x069908A8,0x3D7FEDC4,0x826D2BEF,0x4EEB8476,
0x488DCF25,0x36C9D566,0x28E74E41,0xC2610ACA,0x3D49A9CF,0xBAE3B9DF,
0xB65F8DE6,0x92AEAF64,0x3AC7D5E6,0x9EA80509,0xF22B017D,0xA4173F70,
0xDD1E16C3,0x15E0D7F9,0x50B1B887,0x2B9F4FD5,0x625ABA82,0x6A017962,
0x2EC01B9C,0x15488AA9,0xD716E740,0x40055A2C,0x93D29A22,0xE32DBF9A,
0x058745B9,0x3453DC1E,0xD699296E,0x496CFF6F,0x1C9F4986,0xDFE2ED07,
0xB87242D1,0x19DE7EAE,0x053E561A,0x15AD6F8C,0x66626C1C,0x7154C24C,
0xEA082B2A,0x93EB2939,0x17DCB0F0,0x58D4F2AE,0x9EA294FB,0x52CF564C,
0x9883FE66,0x2EC40581,0x763953C3,0x01D6692E,0xD3A0C108,0xA1E7160E,
0xE4F2DFA6,0x693ED285,0x74904698,0x4C2B0EDD,0x4F757656,0x5D393378,
0xA132234F,0x3D321C5D,0xC3F5E194,0x4B269301,0xC79F022F,0x3C997E7E,
0x5E4F9504,0x3FFAFBBD,0x76F7AD0E,0x296693F4,0x3D1FCE6F,0xC61E45BE,
0xD3B5AB34,0xF72BF9B7,0x1B0434C0,0x4E72B567,0x5592A33D,0xB5229301,
0xCFD2A87F,0x60AEB767,0x1814386B,0x30BCC33D,0x38A0C07D,0xFD1606F2,
0xC363519B,0x589DD390,0x5479F8E6,0x1CB8D647,0x97FD61A9,0xEA7759F4,
0x2D57539D,0x569A58CF,0xE84E63AD,0x462E1B78,0x6580F87E,0xF3817914,
0x91DA55F4,0x40A230F3,0xD1988F35,0xB6E318D2,0x3FFA50BC,0x3D40F021,
0xC3C0BDAE,0x4958C24C,0x518F36B2,0x84B1D370,0x0FEDCE83,0x878DDADA,
0xF2A279C7,0x94E01BE8,0x90716F4B,0x954B8AA3},          //S[6]

{0xE216300D,0xBBDDFFFC,0xA7EBDABD,0x35648095,0x7789F8B7,0xE6C1121B,
```

```
0x0E241600,0x052CE8B5,0x11A9CFB0,0xE5952F11,0xECE7990A,0x9386D174,
0x2A42931C,0x76E38111,0xB12DEF3A,0x37DDDDFC,0xDE9ADEB1,0x0A0CC32C,
0xBE197029,0x84A00940,0xBB243A0F,0xB4D137CF,0xB44E79F0,0x049EEDFD,
0x0B15A15D,0x480D3168,0x8BBBDE5A,0x669DED42,0xC7ECE831,0x3F8F95E7,
0x72DF191B,0x7580330D,0x94074251,0x5C7DCDFA,0xABBE6D63,0xAA402164,
0xB301D40A,0x02E7D1CA,0x53571DAE,0x7A3182A2,0x12A8DDEC,0xFDAA335D,
0x176F43E8,0x71FB46D4,0x38129022,0xCE949AD4,0xB84769AD,0x965BD862,
0x82F3D055,0x66FB9767,0x15B80B4E,0x1D5B47A0,0x4CFDE06F,0xC28EC4B8,
0x57E8726E,0x647A78FC,0x99865D44,0x608BD593,0x6C200E03,0x39DC5FF6,
0x5D0B00A3,0xAE63AFF2,0x7E8BD632,0x70108C0C,0xBBD35049,0x2998DF04,
0x980CF42A,0x9B6DF491,0x9E7EDD53,0x06918548,0x58CB7E07,0x3B74EF2E,
0x522FFFB1,0xD24708CC,0x1C7E27CD,0xA4EB215B,0x3CF1D2E2,0x19B47A38,
0x424F7618,0x35856039,0x9D17DEE7,0x27EB35E6,0xC9AFF67B,0x36BAF5B8,
0x09C467CD,0xC18910B1,0xE11DBF7B,0x06CD1AF8,0x7170C608,0x2D5E3354,
0xD4DE495A,0x64C6D006,0xBCC0C62C,0x3DD00DB3,0x708F8F34,0x77D51B42,
0x264F620F,0x24B8D2BF,0x15C1B79E,0x46A52564,0xF8D7E54E,0x3E378160,
0x7895CDA5,0x859C15A5,0xE6459788,0xC37BC75F,0xDB07BA0C,0x0676A3AB,
0x7F229B1E,0x31842E7B,0x24259FD7,0xF8BEF472,0x835FFCB8,0x6DF4C1F2,
0x96F5B195,0xFD0AF0FC,0xB0FE134C,0xE2506D3D,0x4F9B12EA,0xF215F225,
0xA223736F,0x9FB4C428,0x25D04979,0x34C713F8,0xC4618187,0xEA7A6E98,
0x7CD16EFC,0x1436876C,0xF1544107,0xBEDEEE14,0x56E9AF27,0xA04AA441,
0x3CF7C899,0x92ECBAE6,0xDD67016D,0x151682EB,0xA842EEDF,0xFDBA60B4,
0xF1907B75,0x20E3030F,0x24D8C29E,0xE139673B,0xEFA63FB8,0x71873054,
0xB6F2CF3B,0x9F326442,0xCB15A4CC,0xB01A4504,0xF1E47D8D,0x844A1BE5,
0xBAE7DFDC,0x42CBDA70,0xCD7DAE0A,0x57E85B7A,0xD53F5AF6,0x20CF4D8C,
0xCEA4D428,0x79D130A4,0x3486EBFB,0x33D3CDDC,0x77853B53,0x37EFFCB5,
0xC5068778,0xE580B3E6,0x4E68B8F4,0xC5C8B37E,0x0D809EA2,0x398FEB7C,
0x132A4F94,0x43B7950E,0x2FEE7D1C,0x223613BD,0xDD06CAA2,0x37DF932B,
0xC4248289,0xACF3EBC3,0x5715F6B7,0xEF3478DD,0xF267616F,0xC148CBE4,
0x9052815E,0x5E410FAB,0xB48A2465,0x2EDA7FA4,0xE87B40E4,0xE98EA084,
0x5889E9E1,0xEFD390FC,0xDD07D35B,0xDB485694,0x38D7E5B2,0x57720101,
0x730EDEBC,0x5B643113,0x94917E4F,0x503C2FBA,0x646F1282,0x7523D24A,
0xE0779695,0xF9C17A8F,0x7A5B2121,0xD187B896,0x29263A4D,0xBA510CDF,
0x81F47C9F,0xAD1163ED,0xEA7B5965,0x1A00726E,0x11403092,0x00DA6D77,
0x4A0CDD61,0xAD1F4603,0x605BDFB0,0x9EEDC364,0x22EBE6A8,0xCEE7D28A,
0xA0E736A0,0x5564A6B9,0x10853209,0xC7EB8F37,0x2DE705CA,0x8951570F,
0xDF09822B,0xBD691A6C,0xAA12E4F2,0x87451C0F,0xE0F6A27A,0x3ADA4819,
0x4CF1764F,0x0D771C2B,0x67CDB156,0x350D8384,0x5938FA0F,0x42399EF3,
0x36997B07,0x0E84093D,0x4AA93E61,0x8360D87B,0x1FA98B0C,0x1149382C,
0xE97625A5,0x0614D1B7,0x0E25244B,0x0C768347,0x589E8D82,0x0D2059D1,
0xA466BB1E,0xF8DA0A82,0x04F19130,0xBA6E4EC0,0x99265164,0x1EE7230D,
0x50B2AD80,0xEAEE6801,0x8DB2A283,0xEA8BF59E}};            //S[7]
```

8.3 习题与实践题

8.3.1 习题

1. 简要说明 CAST-128 加密算法的轮运算中 F 函数运算过程的基本原理。
2. 试描述 CAST-128 加密算法的加密和解密过程。

8.3.2 实践题

编程实现 CAST-128 加密算法,要求:待加密的消息从文件中读取,加密后的消息存储到文件,经过解密后得到的消息存储到相应的文件,并比较加密前的消息和加密后的消息,判断是否正确解密消息。

第 9 章 分组密码模式

分组密码是在明文分组和密文分组上进行运算,如果对相同的明文分组进行加密得到的密文分组也将完全相同。

密码模式通常由基本密码、反馈或其他一些简单的运算组成。一些密码模式起到隐藏明文和密文关系的作用,但加密过程的安全性主要依赖于加密算法的安全性,同时密码模式不能破坏加密过程的安全性。在选择密码模式时还需要考虑密码模式的效率,密码模式至少不能明显地降低加密和解密的效率。

9.1 电子密码本模式

电子密码本模式(Electronic Code Block,ECB)又称为电码本模式,在加密过程中将明文分割成相同长度的分组,然后进行加密。解密过程与加密过程的方法相同。电码本的加密和解密的具体过程如图 9-1 所示。

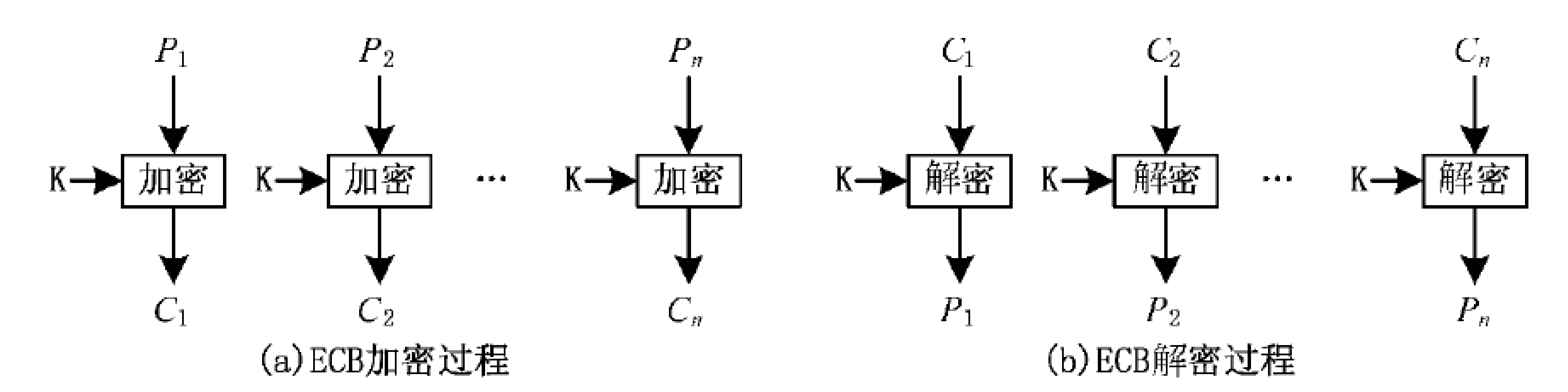


图 9-1 电码本工作模式

电码本的加密和解密的工作模式也可以描述为

$$\begin{cases} C_i = E_k(P_i) \\ P_i = D_k(C_i) \end{cases} \quad (i = 1,2,\cdots,n;n \text{ 为分组数目}) \tag{9-1}$$

在电码本加密过程中,大多数消息的长度并不是分组长度的整数倍,通常最后一个分组为短分组,因此常常需要对最后一个分组进行填充以达到分组的长度。一般的填充方法是填充 0 或 1,或 0 和 1 交替填充,把最后一个分组填充为一个完整的分组。也可以在填充前加一个结束标志,然后再进行填充,使用这种填充方法在解密的时候方便将填充的数据删除,以达到正常还原数据的目的。

电码本加密模式简单有效,方便进行并行加密。但它不能隐藏明文的加密信息,相同的明文总是得到相同的密文,在传递太多的信息时,容易给攻击者提供攻击的机会。同时,在

传递密文中，如果密文的某个分组的位数发生变化时会影响其后密文的解密。因此，电码本模式比较适合于传输短消息。

由于电码本模式的加密和解密过程都是针对各分组单独进行，因此电码本模式的加密和解密的过程都可以采用并行计算的方法进行。

9.2 密码分组链接模式

在密码分组链接模式(Cipher Block Chaining,CBC)中引入了反馈机制，每组密文不仅与当前的明文相关，还与前面的各组明文或生成的密文相关。每一组的分组加密结果反馈到下一组的加密，使得每一组的密文不仅依赖于当前的明文，还依赖于上一组的密文，有效克服了电码本模式中相同明文分组加密得到相同密文分组的缺点。密码分组链接模式的加密和解密过程如图 9-2 所示。

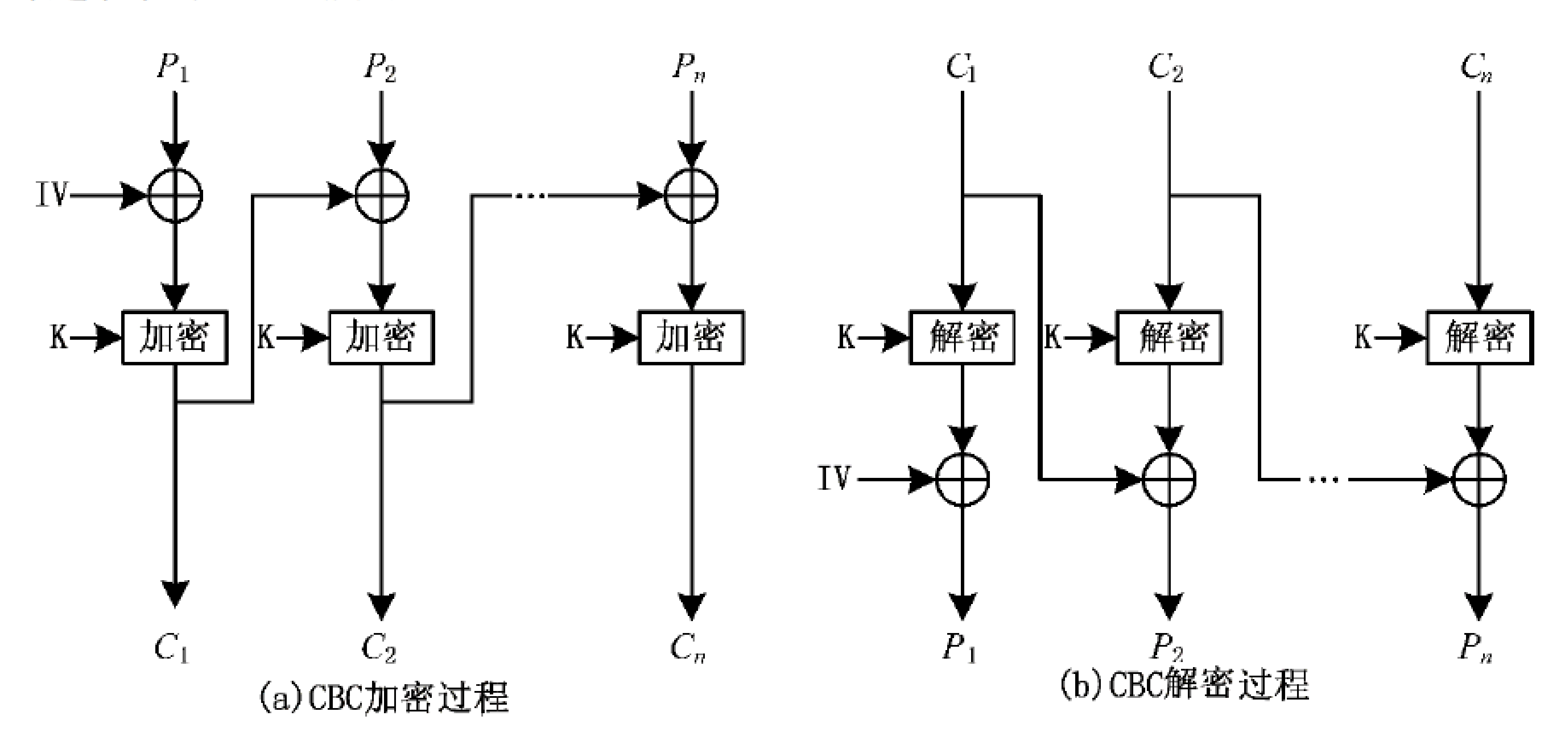


图 9-2 密码分组链接模式的加密和解密过程

密码分组链接模式的加密过程可以描述为

$$C_1 = E_k(P_1 \oplus \mathrm{IV})$$
$$C_i = E_k(C_{i-1} \oplus P_i) \quad (i = 2,3,\cdots,n) \tag{9-2}$$

如果分组起始的索引为 1，那么密码分组链接模式的加密过程可以描述为

$$C_i = E_k(C_{i-1} \oplus P_i), \quad C_0 = \mathrm{IV} \tag{9-3}$$

密码分组链接模式的解密过程可以描述为

$$P_1 = D_k(C_1) \oplus \mathrm{IV}$$
$$P_i = D_k(C_i) \oplus C_{i-1} \quad (i = 2,3,\cdots,n) \tag{9-4}$$

如果分组起始的索引为 1，那么密码分组链接模式的解密过程可以描述为

$$P_i = D_k(C_i) \oplus C_{i-1}, \quad C_0 = \mathrm{IV} \tag{9-5}$$

在密码分组链接模式中，当一组密文发生错误时会影响下一组密文的解密，但再接下来一组密文的解密不会受到影响。但在密文中丢失位数时，若不是丢失整个分组的位数，那么后续所有的密文组都会受到影响，因此在使用过程中需确保整个密码结构的完整。

由于在加密过程中前一分组和后一分组之间存在关联，现在还没有合适的方式进行并

行计算。但由于在解密过程中一个分组的错误只会影响下一分组的解密，而不会影响再下一分组的解密，因此在解密过程中可以使用并行的方法进行解密。

9.3 明文密码分组链接模式

明文密码分组链接模式(Plaintext Cipher-block Chaining，PCBC)又称为传播密码分组链接模式(Propagating Cipher-block Chaining)，是在密码分组链接模式上发展而来的。明文密码分组链接模式是将明文和密文异或后用于下一组明文的加密，明文密码分组链接模式的加密和解密过程如图9-3所示。

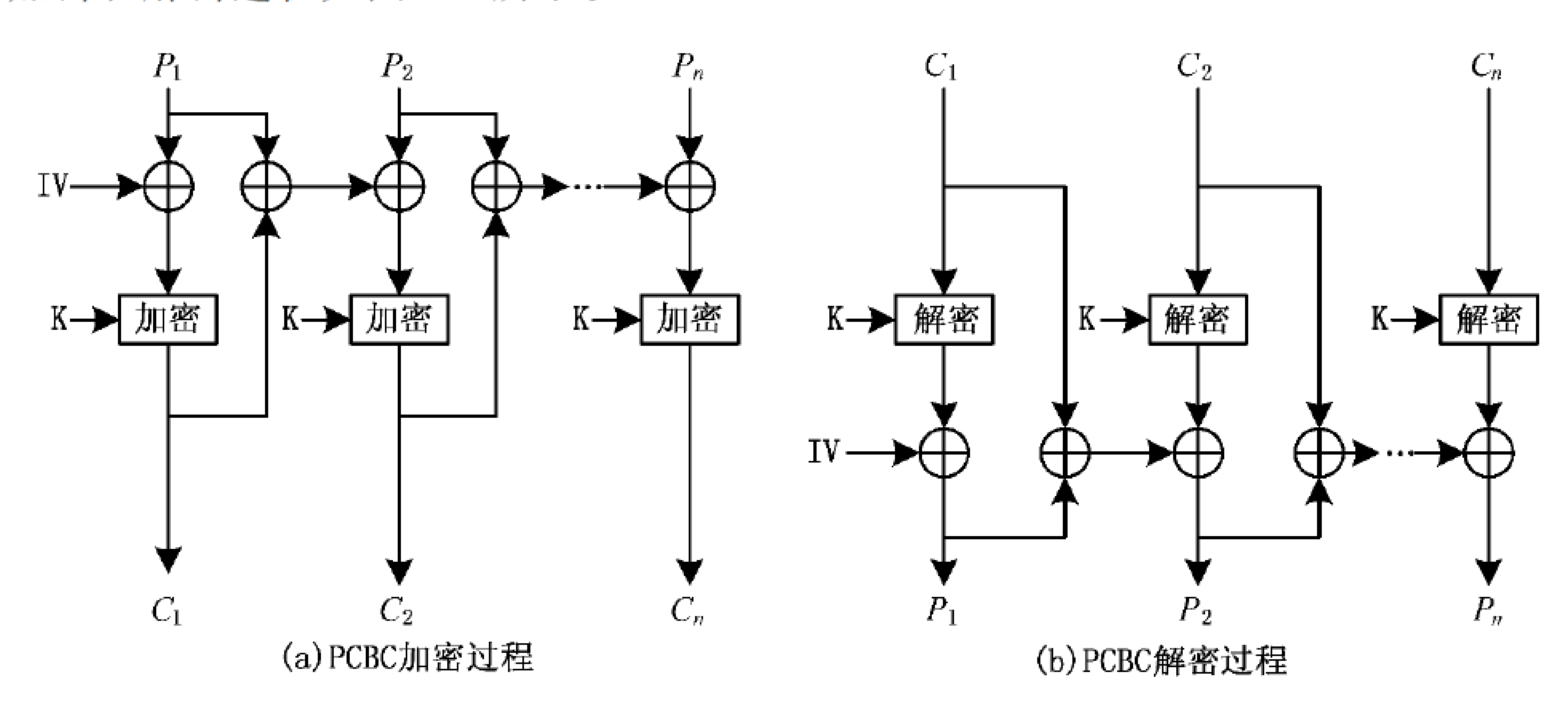


图9-3 明文密码分组链接模式

如果分组起始的索引为1，那么明文密码分组链接模式的加密和解密过程可以描述为

$$\begin{cases} C_i = E_k(P_i \oplus P_{i-1} \oplus C_{i-1}) \\ P_i = D_k(C_i) \oplus P_{i-1} \oplus C_{i-1} \end{cases}, \quad P_0 \oplus C_0 = \mathrm{IV} \tag{9-6}$$

在明文密码分组链接模式中，由于加密和解密过程中均用到上一组的明文和密文，因此明文密码分组链接模式的加密和解密均不能常用并行计算的方法。明文密码分组链接模式目前主要用于Kerberos v4网络协议中。

9.4 密码反馈模式

密码反馈模式(Cipher Feedback，CFB)是与密码分组链接模式最为接近的一种加密模式，在加密过程中将分组加密为一密钥流，其加密的方法就如同倒置的密码分组链接模式，按照分组长度的简化密码反馈模式的加密和解密过程如图9-4所示。

如果分组起始索引为1，那么密码反馈模式的加密和解密过程可以描述为

$$\begin{cases} C_i = E_k(C_{i-1}) \oplus P_i \\ P_i = E_k(C_{i-1}) \oplus C_i \end{cases}, \quad C_0 = \mathrm{IV} \tag{9-7}$$

密码反馈模式不仅可以按照分组长度大小进行加密和解密，还可以按照比分组长度更小的长度进行加密，例如：1-bit CFB、8-bit CFB、64-bit CFB、128-bit CFB等。

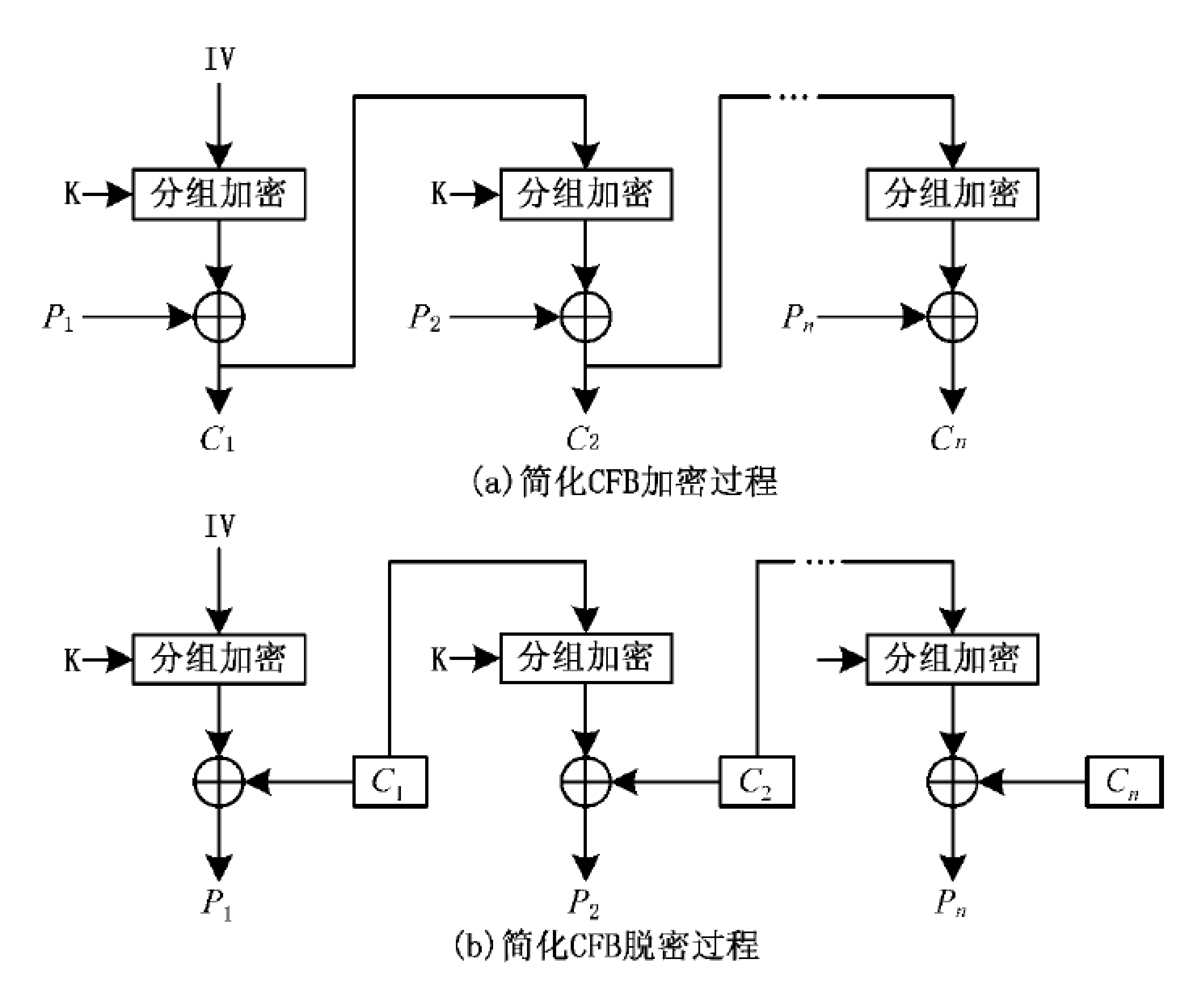


图 9-4 简化密码反馈模式的加密和解密过程

假设明文的分组长度为 s 位，加密分组长度为 b 位，初始化 IV 的长度为 b 位，密码反馈模式的加密和解密过程如图 9-5 所示。

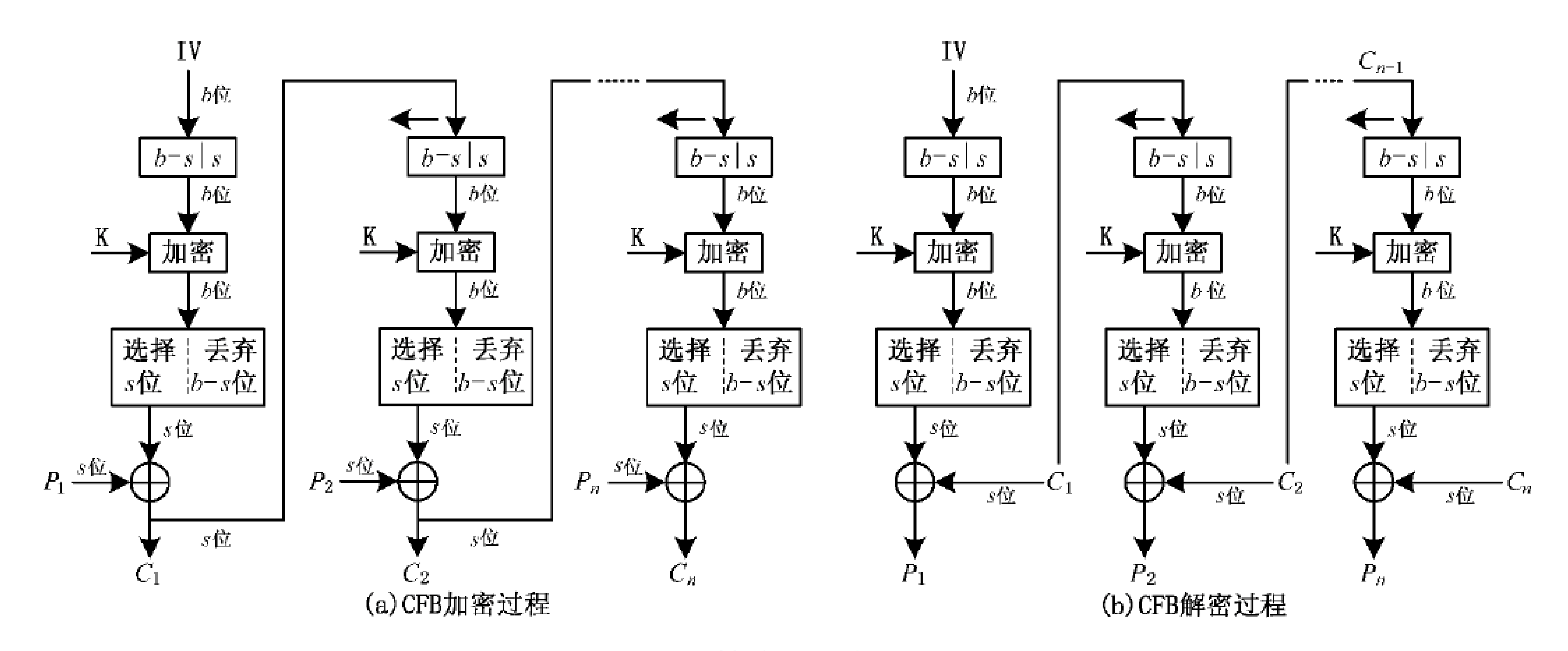


图 9-5 密码反馈模式的加密和解密过程

密码反馈模式的加密过程可以描述为

$$\begin{cases} I_1 = \mathrm{IV} \\ I_i = \mathrm{LSB}_{b-s}(I_{i-1}) \mid\mid C_{i-1}, & i = 2,3,\cdots,n \\ O_i = E_k(I_i), & i = 1,2,\cdots,n \\ C_i = P_i \oplus \mathrm{MSB}_s(O_i), & i = 1,2,\cdots,n \end{cases} \tag{9-8}$$

密码反馈模式的解密过程可以描述为

$$\begin{cases} I_1 = \mathrm{IV} \\ I_i = \mathrm{LSB}_{b-s}(I_{i-1}) \,||\, C_{i-1}, & i = 2,3,\cdots,n \\ O_i = E_k(I_i), & i = 1,2,\cdots,n \\ P_i = C_i \oplus \mathrm{MSB}_s(O_i), & i = 1,2,\cdots,n \end{cases} \tag{9-9}$$

式中：IV 表示初始向量，LSB 表示最低有效位，MSB 表示最高有效位，||表示连接。

密码反馈模式的加密过程如下：

(1) 将 b 位 IV 初始向量存入移位寄存器，作为下一步的输入。

(2) 对输入 b 位数据进行加密，输出 b 位数据。

(3) 取最高有效位 s 位数据与明文进行异或运算，得到密文。

(4) 将移位寄存器左移 s 位，将步骤(3)得到的密文填充到移位寄存器的最低有效位 s 位。

(5) 重复步骤(2)到(4)直到所有明文加密完毕。

密码反馈模式的解密过程如下：

(1) 将 b 位 IV 初始向量存入移位寄存器，作为下一步的输入。

(2) 对输入 b 位数据进行加密，输出 b 位数据。

(3) 取最高有效位 s 位数据与密文进行异或运算，得到明文。

(4) 将移位寄存器左移 s 位，使用步骤(3)得到的密文填充到移位寄存器的最低有效位 s 位。

(5) 重复步骤(2)到(4)直到所有密文解密完毕。

密码反馈模式有效隐藏了明文的加密模式，但由于在加密过程中使用了上一步加密得到的密文作为移位寄存器的输入，因此密码反馈模式目前尚无合适的并行计算方法。密码反馈模式在解密过程中 1 位的错误仅会对 $b/s+1$ 个密文的解密产生影响，因此密码反馈模式可以采用并行计算的方法进行解密。

9.5 输出反馈模式

输出反馈模式(Output Feedback，OFB)的加密方式与密码反馈模式的加密过程类似，在加密的过程中是将加密后的数据作为反馈输入到下一步，而不是将加密后的密文作为下一步的输入。输出反馈模式的加密过程中也需要初始向量 IV 作为第一步的输入，输出反馈模式完整的加密和解密过程如图 9-6 所示。

输出反馈模式的加密过程可以描述为

$$\begin{cases} I_1 = \mathrm{IV} \\ I_i = O_{i-1}, & i = 2,3,\cdots,n \\ O_i = E_k(I_i), & i = 1,2,\cdots,n \\ C_i = P_i \oplus O_i, & i = 1,2,\cdots,n-1 \\ C_n = P_n \oplus \mathrm{MSB}_s(O_n) \end{cases} \tag{9-10}$$

输出反馈模式的解密过程可以描述为

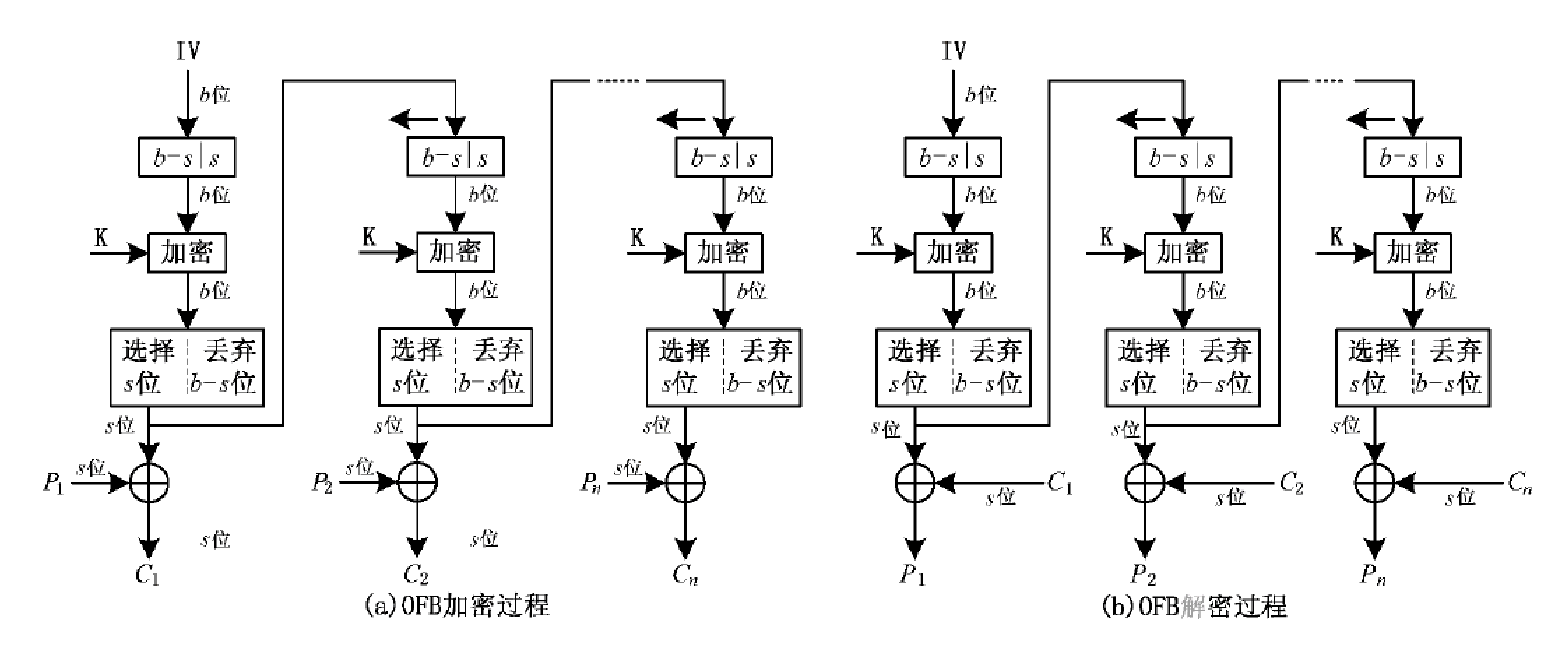


图 9-6 输出反馈模式的加密和解密过程

$$\begin{cases} I_1 = \mathrm{IV} & \\ I_i = O_{i-1}, & i = 2,3,\cdots,n \\ O_i = E_k(I_i), & i = 1,2,\cdots,n \\ P_i = C_i \oplus O_i, & i = 1,2,\cdots,n-1 \\ P_n = C_n \oplus \mathrm{MSB}_s(O_n), & \end{cases} \tag{9-11}$$

输出反馈模式的加密过程如下：

(1) 将 b 位 s 初始向量存入移位寄存器，作为下一步的输入。

(2) 对输入 b 位数据进行加密，输出 b 位数据。

(3) 取最高有效位 s 位数据与明文进行异或运算，得到密文。

(4) 将移位寄存器左移 s 位，将步骤(2)得到的输出取最高有效位 s 位填充到移位寄存器的最低有效位 s 位。

(5) 重复步骤(2)到(4)直到所有明文加密完毕。

输出反馈模式的解密过程如下：

(1) 将 b 位 s 初始向量存入移位寄存器，作为下一步的输入。

(2) 对输入 b 位数据进行加密，输出 b 位数据。

(3) 取最高有效位 s 位数据与密文进行异或运算，得到明文。

(4) 将移位寄存器左移 s 位，将步骤(2)得到的输出取最高有效位 s 位填充到移位寄存器的最低有效位 s 位。

(5) 重复步骤(2)～步骤(4)直到所有密文解密完毕。

输出反馈模式的加密和解密过程到目前为止尚无合适的并行计算方法。

9.6 计数器模式

计数器模式(Counter,CTR)是将一系列计数器的值作为输入应用到加密过程中，将该值使用密钥进行加密后与明文进行异或得到密文。计数器模式的加密是针对计数器的值进行加密，计数器模式的加密和解密的过程如图 9-7 所示。

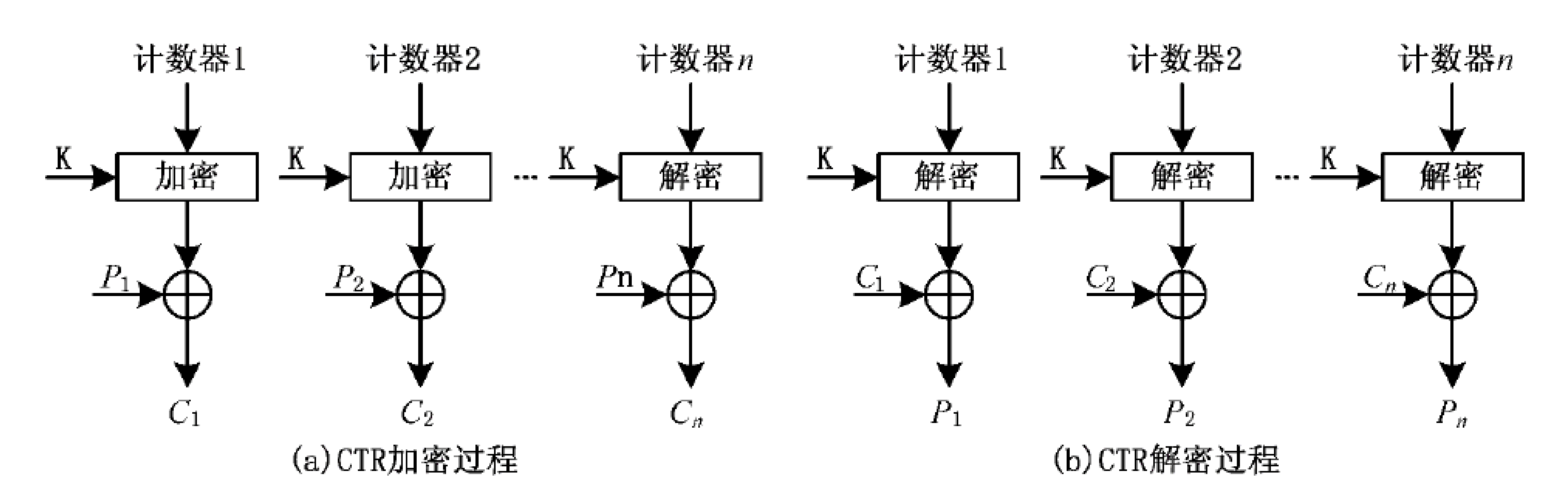


图 9-7 计数器模式的加密和解密过程

计数器模式的加密过程可以描述为

$$\begin{cases} O_i = E_k(T_i), & i = 1,2,\cdots,n \\ C_i = P_i \oplus O_i, & i = 1,2,\cdots,n-1 \\ C_n = P_n \oplus \mathrm{MSB}_s(O_n), \end{cases} \tag{9-12}$$

计数器模式的解密过程可以描述为

$$\begin{cases} O_i = E_k(T_i), & i = 1,2,\cdots,n \\ P_i = C_i \oplus O_i, & i = 1,2,\cdots,n-1 \\ P_n = C_n \oplus \mathrm{MSB}_s(O_n) \end{cases} \tag{9-13}$$

其中：T_i 是计数器序列，MSB 是指最高有效位。

对于加密过程的最后一个分组，其长度可能小于分组的长度，因此针对最后一个分组的特点，在加密过程中将最后一个计数器的解密结果取出与最后一个分组长度相等的数据即可，与明文异或后得到相同长度的密文。在计数器模式的解密过程中最后一组数据的处理方法与加密过程中最后一组数据的处理方法相同。

根据计数器模式的加密和解密方式可以看出，计数器模式的加密和解密过程均适合于并行计算。

9.7 填充

在大多数的加密模式中都存在数据填充的问题，通过数据填充可以将明文分割为任意长度的分组，以适应相应的加密算法。

数据填充的方法有很多种，最简单的方法是在数据的尾部添加 1，然后将其余不足的部分全部添加 0。

例如，有 32 位的分组，最后一个分组的消息长度为 23 位，那么这个消息的填充过程如下：

… | 1011 1001 1101 0100 0010 0111 **0000 0000** |

其中：最后 9 位为填充的数据，最后第 9 位填充 1，其余填充 0。

美国国家标准学会(American National Standards Institute，ANSI)提出的填充标准 ANSI X.923 的填充方法为：在最后一个分组的最后一个字节填充需要填充的字节量，而在其余的位置填充 00。

例如,以下为 8 个字节大小的分组,DD 表示数据,最后一个字节 04 表示总共需要填充 4 个字节的数据,其余填充 00。其中 00、04 等为十六进制数。

… | DD DD DD DD DD DD DD DD | DD DD DD DD 00 00 00 04 |

国际标准化组织(International Organization for Standardization, ISO)提出的填充标准 ISO 10126 的填充方法为:在最后一个分组的最后一个字节填充需要填充的字节量,而在其余的位置填充随机字节。

例如,以下为 8 个字节大小的分组,DD 表示数据,最后一个字节 04 表示总共需要填充 4 个字节的数据,其余 3 个字节为填充随机数。其中数据的表示为十六进制数。

… | DD DD DD DD DD DD DD DD | DD DD DD DD 81 A6 23 04 |

消息加密的语法标准 PKCS7 提出的填充方法为:根据需要填充的字节数来进行填充,若需要填充 5 个字节,那么在最后一个需要填充分组的尾部填充 5 个 05(十六进制),具体的填充方法为如下之一:

01

02 02

03 03 03

04 04 04 04

05 05 05 05 05

……

例如,以下为 8 个字节大小的分组,在最后一个分组中需要填充 4 个字节的数据,那么填充的结果如下:

… | DD DD DD DD DD DD DD DD | DD DD DD DD 04 04 04 04 |

最后 4 个字节的填充数据均为 04。

国际标准化组织和国际电工委员会(ISO/IEC)在 2005 年制定的 ISO/IEC 7816-4 标准中规定:在需要填充的分组的尾部字节填充 80(十六进制),后续部分全部填充 00(十六进制)。

例如,以下为 8 个字节大小的分组,在最后一个分组中需要填充 4 个字节的数据,那么填充的结果如下:

… | DD DD DD DD DD DD DD DD | DD DD DD DD 80 00 00 00 |

如果只有一个字节需要填充,那么填充的方法如下:

… | DD DD DD DD DD DD DD DD | DD DD DD DD DD DD DD 80 |

除以上所述填充方法之外,国际标准化组织和国际电工委员会还制定用于 HASH 和 MACs 的 ISO/IEC 10118-1 和 ISO/IEC 9797-1 的标准,该标准是在需要填充的分组后面全部填充 00。

例如,以下为 8 个字节大小的分组,在最后一个分组中需要填充 4 个字节的数据,那么填充的结果如下:

… | DD DD DD DD DD DD DD DD | DD DD DD DD 00 00 00 00 |

全“00”的填充方法存在一定的缺陷,即:不能明确区分分组数据和填充数据,因此其使用范围有相当的局限性。

9.8 习题与实践题

9.8.1 习题

1. 简要说明电子码本模式的基本工作原理。
2. 简要说明密码分组链接模式的基本工作原理。
3. 简要说明明文密码分组链接模式的基本工作原理。
4. 简要说明密码反馈模式的基本工作原理。
5. 简要说明输出反馈模式的基本工作原理。
6. 简要说明计数器模式的基本工作原理。

9.8.2 实践题

1. 在加密过程中经常会遇到待加密消息的长度不是加密方式所需长度的整数倍，此时需要对待加密的消息进行填充，现使用最简单的填充算法，即在消息的最后附加“1”，而其余部分附加“0”的方法对数据进行填充，假设输入的消息不足 64 位，试采用上述的填充算法对消息进行填充，再使用 DES 加密算法对填充后的数据进行加密，并对加密后的数据进行解密。

提示：对加密后的数据需还原至明文，此时需先去掉附加的填充数据。否则解密后的文件将与原消息不同。

2. 对 1 题中的加密过程进行进一步改造，要求采用密码分组链接模式对输入的任意长度的消息进行加密，使用的加密方法仍然为 DES 加密算法。

3. 美国国家标准学会提出的填充标准是在最后一位填充总共填充数据字节数，其余部分填充“0”，试采用该填充方法对待加密的消息进行填充，并使用 AES 加密算法进行加密和解密，输入数据的长度小于分组的长度。

4. 使用 PKCS7 提出的填充算法对待加密的数据进行填充，使用的加密模式为密码反馈模式，使用的加密方法为 IDEA 加密算法。试完成该程序。

提示：当待加密的数据正好是分组长度的整数倍时仍然需要填充，此时填充的字节数正好是分组的长度。

第10章

A5 算 法

A5 算法是欧洲 GMS(Group Special Mobile)标准中规定的加密算法，用于数字蜂窝移动电话的加密。A5 算法属于典型的 LFSR(线性反馈移位寄存器)序列密码算法或流密码算法，目前主要有两个版本：强安全性的 A5/1 算法和弱安全性的 A5/2 算法。

10.1 序列密码原理

10.1.1 基本原理

现代对称密码主要包括分组密码和序列密码。分组密码是将明文分为固定的长度，而每一组是使用同一个密钥来进行加密变换。而序列密码则是采用序列密钥来加密明文序列。例如，存在明文序列 $m=m_0m_1\cdots m_n$ 和 $k=s_0s_1\cdots s_n$，对于明文符号 m_i 采用密钥 s_i 进行加密，其加密过程可以描述为

$$E_k(m)=E_{s_0}(m_0)E_{s_1}(m_1)\cdots E_{s_n}(m_n)=c_0c_1\cdots c_n \tag{10-1}$$

序列密码算法可以分为同步序列密码算法和异步序列密码算法，如图 10-1 所示。

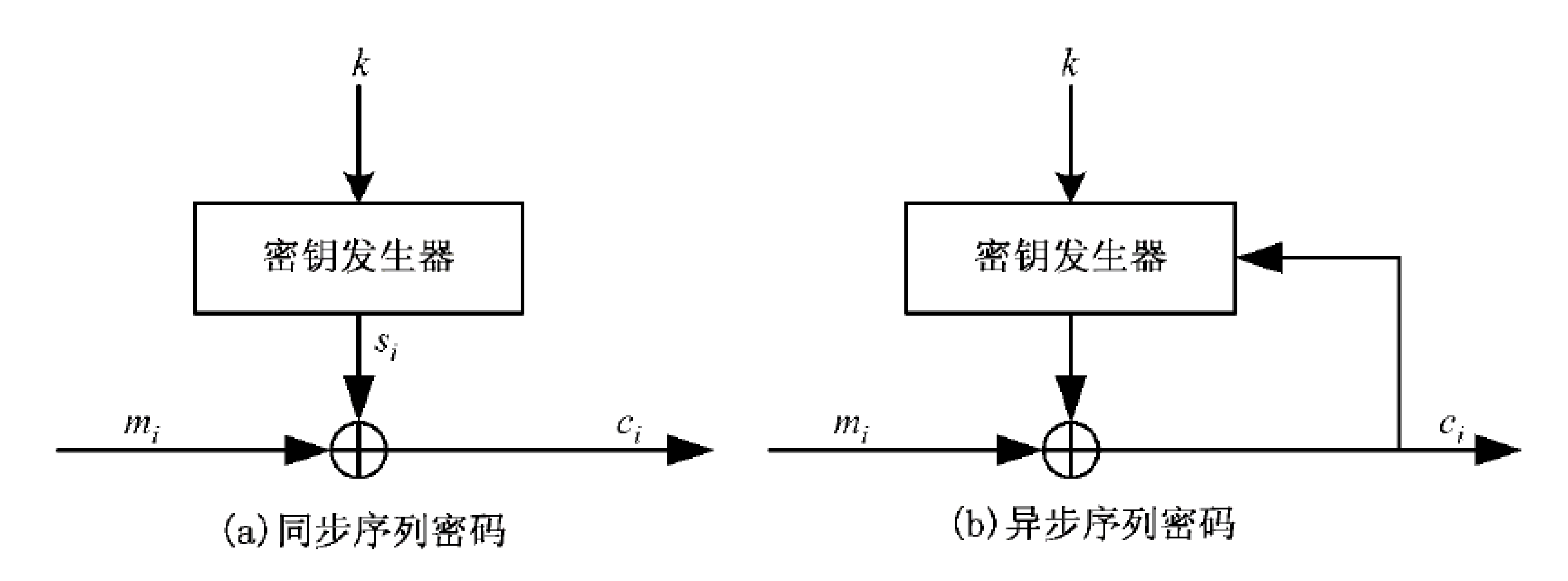


图 10-1 序列密码算法示意图

同步序列密码算法仅依赖于输入的密钥 k，而异步序列密码算法不仅依赖于输入的密钥，同时还依赖于输出的密文。

序列密码算法的解密和解密过程都采用 XOR 操作，并且都是针对单独的位进行运算。加密和解密的算法相同，计算方法见式(9-2)：

$$\begin{aligned} c_i &= E_{s_i}(m_i)\equiv(m_i+s_i)\bmod 2 \\ m_i &= D_{s_i}(c_i)\equiv(c_i+s_i)\bmod 2 \end{aligned} \tag{10-2}$$

加密和解密的转换过程如下：

$$D_{s_i}(c_i) \equiv (c_i + s_i) \bmod 2 \equiv (m_i + s_i + s_i) \bmod 2 \\ \equiv (m_i + 2s_i) \bmod 2 \equiv m_i \bmod 2 \qquad (10\text{-}3)$$

示例 10-1 有 ASCII 编码表的字母“B”,字母“B”的二进制编码为 01000010,假设加密密钥为 00101100,那么加密和解密的过程如下:

$$\begin{array}{lcl} m_0 \cdots m_7 = 01000010 & & c_0 \cdots c_7 = 01101110 \\ \oplus & & \oplus \\ s_0 \cdots s_7 = 00101100 & \rightarrow & s_0 \cdots s_7 = 00101100 \\ c_0 \cdots c_7 = 01101110 & & m_0 \cdots m_7 = 01000010 \end{array} \qquad (10\text{-}4)$$

由于序列密码算法的加密和解密过程完全相同,因此序列密码算法的安全性完全取决于密钥流。生成的序列密钥最好是随机序列,使得密码攻击者不容易对密钥流进行攻击。在实际应用中通常使用伪随机数序列生成器来生成伪随机数序列,并用于生成序列密码算法的密钥流。伪随机数生成的基本原理可以用式(10-5)描述:

$$s_0 = \text{seed}, \quad s_{i+1} = f(s_i), \quad i = 0,1,\cdots \qquad (10\text{-}5)$$

一个比较典型的生成方法是线性同余生成法,其生成原理如下:

$$s_0 = \text{seed}, \quad s_{i+1} \equiv as_i + b \bmod m, \quad i = 0,1,\cdots \qquad (10\text{-}6)$$

其中:a,b,m 为常量。

伪随机数生成器生成的随机数越接近真实的随机数越好,目前序列密钥生成中常使用移位寄存器来生成序列密钥,例如在 A5 算法中使用的就是使用线性反馈移位寄存器(Linear Feedback Shift Regisger,LFSR)来生成所需的密钥。

10.1.2 线性反馈移位寄存器

线性移位反馈寄存器主要包括触发器和反馈路径两部分,线性反馈移位寄存器的存储单元的数量一般称为度,包含 m 个触发器的线性反馈移位寄存器一般称为 m 级线性反馈移位寄存器,图 10-2 是一个简单的 5 级线性反馈移位寄存器:

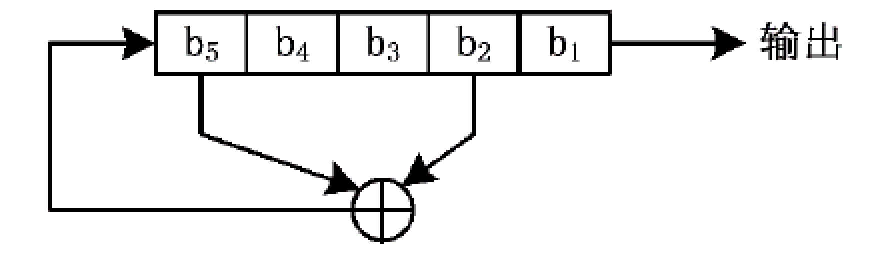


图 10-2 简单的线性反馈移位寄存器

图中的 b_i 表示相应的位,b_2 和 b_5 位置也被称为抽头序列。

示例 10-2 若图 10-5 的初始化值为 11111,那么其运行过程如下:

clk	b_5	b_4	b_3	b_2	b_1
0	1	1	1	1	1
1	0	1	1	1	1
2	1	0	1	1	1
3	0	1	0	1	1
4	1	0	1	0	1
5	1	1	0	1	0
6	0	1	1	0	1
7	0	0	1	1	0
8	1	0	0	1	1
9	0	1	0	0	1
10	0	0	1	0	0

在第 0 次运算中，b_2 和 b_5 进行"异或"运算，得到的结果作为 b_5，b_4 为原 b_5，b_3 为原 b_2。以此类推，最后，b_1 作为输出，其输出的序列为 11111010110…。

示例 10-3 图 10-3(a)是线性反馈移位寄存器的另一种表示方法，它的作用与图 10-3(b)相同。

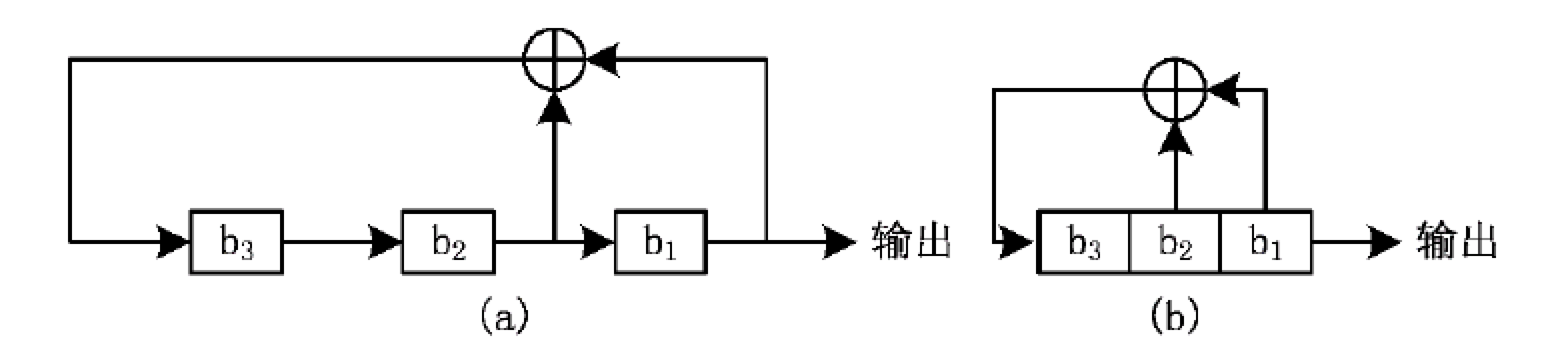


图 10-3 一种 3 级线性反馈移位寄存器

该线性反馈移位寄存器的运行过程如下：

clk	b_3	b_2	b_1
0	1	0	0
1	0	1	0
2	1	0	1
3	1	1	0
4	1	1	1
5	0	1	1
6	0	0	1
7	1	0	0
8	0	1	0

在该线性反馈移位寄存器中，b_2 和 b_1 作为抽头序列，将 b_2 和 b_1"异或"的结果作为 b_3 的输入，b_1 作为输出。

在运行到第 6 个时钟周期之后开始循环，其输出的长度为 7(包含初始值的输出周期)，那么这个线性反馈移位寄存器的输出为

0010111 0010111 0010111 …

通常通过引入多项式来描述线性反馈移位寄存器，见公式(10-7)：

$$P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_1 x + p_0 \tag{10-7}$$

该多项式也被称为关联多项式或联系多项式。

n 位线性反馈移位寄存器的最大输出周期理论上可以达到 2^n，由于全零移位寄存器将循环输出零序列，没有任何意义，因此可能输出的最大周期为 2^n-1，只有特定的抽头序列才能通过所有的 2^n-1 个内部状态，同时抽头序列加参数 1 形成的多项式必须是本原多项式模 2。标记为(5,2,0)的本原多项式是指多项式 x^5+x^2+1，可以用以线性反馈移位寄存器的常用本原多项式见表 10-1。

表 10-1 可用于获得最大输出周期的本原多项式

(2,1,0)	(24,4,3,1,0)	(46,1,0)	(68,7,5,1,0)	(90,5,3,2,0)	(112,5,4,3,0)
(3,1,0)	(25,3,0)	(47,5,0)	(69,6,5,2,0)	(91,8,5,1,0)	(113,5,3,2,0)
(4,1,0)	(26,4,3,1,0)	(48,5,3,2,0)	(70,5,3,1,0)	(92,6,5,2,0)	(114,5,3,2,0)

续表

(5,2,0)	(27,5,2,1,0)	(49,6,5,4,0)	(71,5,3,1,0)	(93,2,0)	(115,8,7,5,0)
(6,1,0)	(28,1,0)	(50,4,3,2,0)	(72,10,9,3,0)	(94,6,5,1,0)	(116,4,2,1,0)
(7,1,0)	(29,2,0)	(51,6,3,1,0)	(73,4,3,2,0)	(95,11,0)	(117,5,2,1,0)
(8,4,3,1,0)	(30,1,0)	(52,3,0)	(74,6,2,1,0)	(96,10,9,6,0)	(118,6,5,2,0)
(9,1,0)	(31,3,0)	(53,6,2,1,0)	(75,6,3,1,0)	(97,6,0)	(119,8,0)
(10,3,0)	(32,7,3,2,0)	(54,8,6,3,0)	(76,5,4,2,0)	(98,7,4,3,0)	(120,4,3,1,0)
(11,2,0)	(33,6,3,1,0)	(55,6,2,1,0)	(77,6,5,2,0)	(99,6,3,1,0)	(121,8,5,1,0)
(12,3,0)	(34,4,3,1,0)	(56,7,4,2,0)	(78,7,2,1,0)	(100,6,5,2,0)	(122,6,2,1,0)
(13,4,3,1,0)	(35,2,0)	(57,4,0)	(79,4,3,2,0)	(101,7,6,1,0)	(123,2,0)
(14,5,0)	(36,5,4,2,0)	(58,6,5,1,0)	(80,9,4,2,0)	(102,6,5,3,0)	(124,37,0)
(15,1,0)	(37,6,4,1,0)	(59,7,4,2,0)	(81,4,0)	(103,9,0)	(125,7,6,5,0)
(16,5,3,1,0)	(38,6,5,1,0)	(60,1,0)	(82,9,6,4,0)	(104,4,3,1,0)	(126,7,4,2,0)
(17,3,0)	(39,4,0)	(61,5,2,1,0)	(83,7,4,2,0)	(105,4,0)	(127,1,0)
(18,3,0)	(40,5,4,3,0)	(62,6,5,3,0)	(84,5,0)	(106,6,5,1,0)	(128,7,2,1,0)
(19,5,2,1,0)	(41,3,0)	(63,1,0)	(85,8,2,1,0)	(107,9,7,4,0)	
(20,3,0)	(42,5,2,1,0)	(64,4,3,1,0)	(86,6,5,2,0)	(108,6,4,1,0)	
(21,2,0)	(43,6,4,3,0)	(65,4,3,1,0)	(87,7,5,1,0)	(109,5,4,2,0)	
(22,1,0)	(44,5,0)	(66,3,0)	(88,11,9,8,0)	(110,6,4,1,0)	
(23,5,0)	(45,4,3,1,0)	(67,5,2,1,0)	(89,6,5,3,0)	(111,7,4,2,0)	

10.2 A5/1 算法原理

A5/1 算法是 A5 系列加密算法中的一种，A5/1 算法的基本加密结构如图 10-4 所示。

A5/1 算法的加密过程非常简单，只需要将输入的明文与密钥流进行异或运算便可以得到密文，因此，A5/1 算法的核心是密钥流的生成，密钥流的生成通过三个线性反馈移位寄存器来完成，三个线性反馈移位寄存器的特征多项分别为

$$\begin{cases} x^{19}+x^5+x^2+x^1+1 \\ x^{22}+x+1 \\ x^{23}+x^{15}+x^2+x+1 \end{cases} \tag{10-8}$$

其密钥流生成的基本过程如图 10-5 所示。

明文 (114+114b) 异或 密文 (114+114b) 伪随机数(228b) Fn(22b) Kc(64b) 密钥序列生成

图 10-4 A5/1 加密算法的基本结构

A5/1 移位寄存器的基本参数见表 10-2。

表 10-2 A5/1 移位寄存器基本参数

编号	长度/b	反馈多项式	时钟位	抽头位
1	19	$x^{19}+x^5+x^2+x^1+1$	8(C1)	13,16,17,18
2	22	$x^{22}+x+1$	10(C2)	20,21
3	23	$x^{23}+x^{15}+x^2+x+1$	10(C3)	7,20,21,22

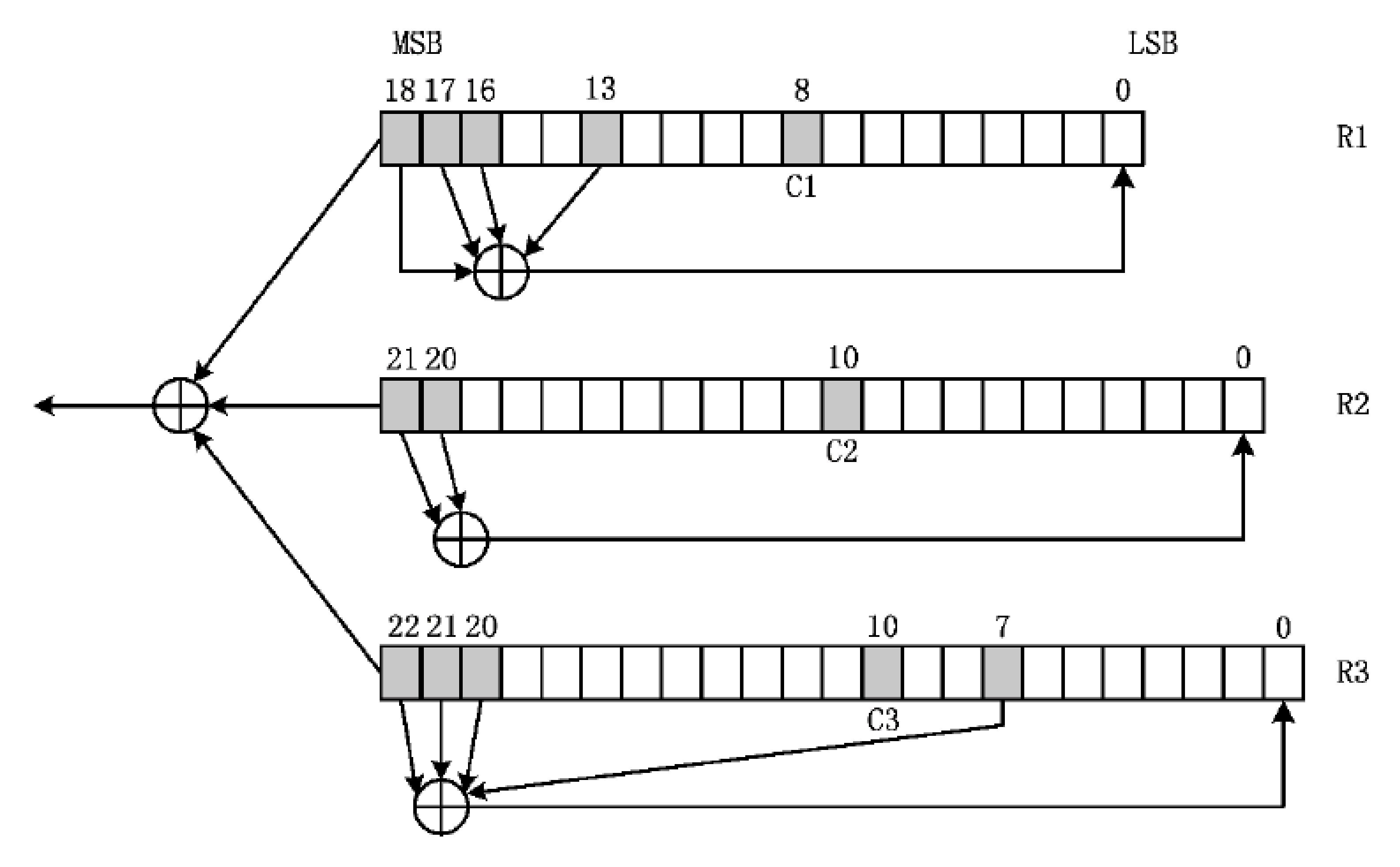


图 10-5 A5/1 密钥流生成结构

A5/1 密钥流的生成过程如下：

(1) 将所有的寄存器置为 0。

(2) 寄存器进行共 64 次的循环(寄存器的总位数为 64 位)，将密钥数据与寄存器混合，对于 $0\leqslant i<64$ 的第 i 次循环，与最低有效位(LSB)进行"异或"运算，$R[0]=R[0]\oplus K[i]$，每次循环处理 1 位数据。

(3) 寄存器再进行 22 次的循环，对于 $0\leqslant i\leqslant 21$ 的第 i 次循环，Fn[i]与寄存器的输入位进行异或运算，并将结果存储到该寄存器的最低有效位，每次循环处理 1 位数据。

(4) 寄存器进行共 100 次的循环(不输出相应数据)，混淆 Kc 和 Fn。完成 100 次的循环之后，寄存器完成初始化。

(5) 寄存器进行 228 次的循环，对于 $0\leqslant i\leqslant 227$ 的第 i 次循环，将 3 个寄存器的最高有效位数据(MSB)进行异或运算，其运算结果为伪随机数的第 i 位。

密钥流生成过程中位状态的变化情况如图 10-6 所示。



图 10-6 密钥流生成中位状态变化

移位寄存器的运行通过钟控位来确定是否运行，若和多数位(Majority)相同则运行，其具体的运行规则如表 10-3 所示。

表 10-3 寄存器运行规则

C_1	C_2	C_3	Majority	R_1	R_2	R_3
0	0	0	0	clock	clock	clock
1	0	0	0		clock	clock
0	1	0	0	clock		clock
1	1	0	1	clock	clock	
0	0	1	0	clock	clock	

续表

C_1	C_2	C_3	Majority	R_1	R_2	R_3
1	0	1	1	clock		clock
0	1	1	1		clock	clock
1	1	1	1	clock	clock	clock

A5/1 的加密和解密算法非常简单，加密的过程是将明文与加密密钥流进行异或运算获得密文，而解密过程则是将密文与解密密钥进行异或运算得到明文。

10.3 A5/1 算法实现

A5/1 算法的实现主要是模拟硬件实现的过程，整个实现过程的核心部分是密钥的生成。密钥的生成过程由以下几部分来完成：

(1) 输入密钥。

(2) 输入 Fn。

(3) 100 时钟周期的运行。

(4) 生成密钥流及加密。

10.3.1 A5/1 算法实现的基本结构

A5/1 算法主要由 A5_1 类来实现相关的功能，A5_1 类的基本结构见图 10-7。A5_1 类的声明见程序清单 10-1。

A5_1	
- R1 : word	
- R2 : word	
- R3 : word	
+ parity (word x)	: bit
+ clockOne (word reg, word mask, word taps)	: word
+ majority ()	: bit
+ clock ()	: void
+ clockAllThree ()	: void
+ getBit ()	: bit
+ setKey (byte key[], word frame)	: void
+ getKey (byte aToBKeyStream[], byte bToAKeyStream[])	: void
+ run ()	: void

图 10-7 A5_1 软件实现的基本结构

程序清单 10-1

```
#define R1MASK 0x07FFFF
#define R2MASK 0x3FFFFF
#define R3MASK 0x7FFFFF
#define R1MID 0x000100
#define R2MID 0x000400
#define R3MID 0x000400
```

```
07 #define R1TAPS 0x072000
08 #define R2TAPS 0x300000
09 #define R3TAPS 0x700080
10 #define R1OUT 0x040000
11 #define R2OUT 0x200000
12 #define R3OUT 0x400000
13 typedef unsigned char byte;
14 typedef unsigned long word;
15 typedef word bit;
16 class A5_1
17 {
18     public:
19         bit parity(word x);
20         word clockOne(word reg,word mask,word taps);
21         bit majority();
22         void clock();
23         void clockAllThree();
24         bit getBit();
25         void setKey(byte key[],word frame);
26         void getKey (byte aToBKeyStream[],byte bToAKeyStream[]);
27         void run();
28     private:
29         word R1,R2,R3;
30 };
```

程序清单 10-1 中的第 1 行到第 12 行为处理 3 个寄存器用的相关声明，表示方法为十六进制。

R1MASK、R2MASK 和 R3MASK 是三个寄存器的掩码，R1 为 19 位，R2 为 22 位，R3 为 23 位，分别对应三个特征多项式，例如：R1MASK 为 0x07FFFF，转换为二进制格式则为 $(000001111111111111111111)_2$，与第一个寄存器对应。

R1MID、R2MID 和 R3MID 用于获得特征多项式钟控位的数据，特征多项式的钟控位从“0”开始计算，则 R1MID、R2MID 和 R3MID 分别对应第一个寄存器的第 8 位、第二个寄存器的第 10 位和第三个寄存器的第 10 位，例如：R1MID 为 0x000100，转换成二进制格式为 $(\cdots 0100000000)_2$，与钟控位位置正好对应。

R1TAPS、R2TAPS 和 R3TAPS 分别对应三个特征多项式的抽头位，例如 R1TAPS 为 0x072000，转换成二进制格式为 $(000001110010000000000000)_2$，那么，抽头位为第 18、17、16 和 13 位。

R1OUT、R2OUT 和 R3OUT 在获得输出数据时使用，分别对应第 18、21 和 22 位，将三个寄存器对应位进行异或计算后作为输出。

byte、word 和 bit 的定义是为了方便后续数据处理。

各函数的主要功能和参数说明如下：

- parity(word x)——判断参数 x 模 2 运算得到结果的奇偶性，函数的返回类型为 bit。

- clockOne(word reg,word mask,word taps)——用于模拟寄存器运行一次，函数返回类型为 word，函数参数为寄存器、寄存器掩码、和寄存器抽头。
- majority()——获得 3 个寄存器钟控位的情况，函数返回类型为 bit。
- clock()——根据钟控位的状态运行寄存器。
- clockAllThree()——同时运行 3 个寄存器。
- getBit()——获取 3 个寄存器的输出。
- setKey()——预处理密钥，为后续获取密钥和加密做准备，函数参数为输入的密钥和 Fn。
- getKey()——获取加密和解密密钥流。
- run()——测试运行加密和解密过程。

10.3.2 A5/1 算法具体实现

A5/1 算法的密钥生成首先要完成输入密钥、Fn，然后再经过 100 个时钟周期的混合过程。以上过程是通过函数 setKey()来完成的。setKey()函数的详细内容见程序清单 10-2。

程序清单 10-2

```
void A5_1::setKey(byte key[],word frame)
{
    int i;
    bit keyBit,frameBit;
    R1=0;
    R2=0;
    R3=0;
    for(i=0;i<64;i++)
    {
        clockAllThree();
        keyBit=(key[i/8]>>(i&7))&1;
        R1^=keyBit;
        R2^=keyBit;
        R3^=keyBit;
    }
    for(i=0;i<22;i++)
    {
        clockAllThree();
        frameBit=(frame>>i)&1;
        R1^=frameBit;
        R2^=frameBit;
        R3^=frameBit;
    }
    for(i=0;i<100;i++)
    {
        clock();
    }
```

```
28  }
```

在函数的实现过程中，首先将寄存器置为 0，然后通过第 8 行到第 15 行的代码将输入密钥初始化，进行 64 次循环操作，再通过第 16 行到第 23 行代码初始化 Fn，进行 22 次循环操作，最后进行 100 次混淆操作。在上述过程中还分别用到了 clockAllThree() 函数和 clock() 函数。这两个函数还用到一些相关的辅助函数，以下逐步介绍各个函数的功能。

parity(word x)用于判断各位数据和模 2 的奇偶性，在 majority() 函数、clock() 等函数中均使用该函数，函数具体内容见程序清单 10-3。

程序清单 10-3

```
bit A5_1::parity(word x)
{
      x^=x>>16;
      x^=x>>8;
      x^=x>>4;
      x^=x>>2;
      x^=x>>1;
      return x&1;
}
```

例如，可以通过该函数来判断钟控位与寄存器当前位的奇偶性，使用 parity(R1&R1MID)时，若 R1 在钟控位为 1 时，R1&R1MID 为奇数则 parity(R1&R1MID)返回 1，否则返回 0。

函数的参数为 word 型 32 位数据，在计算过程中将参数分为两部分，并利用异或运算将对应位置(二进制)都为 1 的数据置为 0，以此类推至处理完所有数据。例如，假设输入的数据为 16 位 X$(1101100111110101)_2$，其计算过程如下：

$(1101100111110101)_2 >> 8 \rightarrow (0000000011011001)_2$

$(\mathbf{11110101})_2$^$(\mathbf{11011001})_2 \rightarrow (\mathbf{00101100})_2$

$(\mathbf{00101100})_2 >> 4 \rightarrow (0000\mathbf{0010})_2$

$(\mathbf{1100})_2$^$(\mathbf{0010})_2 \rightarrow (\mathbf{1110})_2$

$(\mathbf{1110})_2 >> 2 \rightarrow (\mathbf{0011})_2$

$(\mathbf{10})_2$^$(\mathbf{11})_2 \rightarrow (\mathbf{01})_2$

$(\mathbf{01})_2 >> 1 \rightarrow (\mathbf{00})_2$

$(\mathbf{1})_2$^$(\mathbf{0})_2 \rightarrow (\mathbf{1})_2$

最后再与 1 进行“&”运算，将其他位的数据置为 0，完成整个计算。

只需获得最后一次计算结果，因此在计算过程中每次均忽略了左半部分，该部分不会影响计算的结果。

函数 majority()用于判断 3 个寄存器在钟控位 1 为多数还是 0 为多数，当 1 为多数时返回 1，当 0 为多数时返回 0。采用的方法为计算奇数的个数来确定，若奇数的个数大于等于 2 则 1 为多数，返回 1，否则返回 0。具体实现的代码见程序清单 10-4。

程序清单 10-4

```
bit A5_1::majority()
```

```
{
    int sum;
    sum=parity(R1&R1MID)+parity(R2&R2MID)+parity(R3&R3MID);
    if(sum>=2)
    {
        return 1;
    }
    return 0;
}
```

majority()函数通过统计三个移位寄存器在钟控位数据是否是“1”来确定钟控位为“1”的和，例如，R1&R1MID 中的 R1MID 在第 9 位为“1”，如果 R1 的第 9 位为“1”，则运算结果为“1”，通过 parity(R1&R1MID)计算得到结果。分别对三个移位寄存器计算后可得到最终结果，依次可以计算得到三个移位寄存器的钟控位是“0”为多数还是“1”为多数，最终确定移位寄存器的运行状态。

clockOne(word reg，word mask，word taps)函数用于模拟寄存器的运行，运行完成后将结果返回给当前寄存器，具体的实现代码见程序清单 10-5。

程序清单 10-5

```
word A5_1::clockOne(word reg,word mask,word taps)
{
    word t=reg&taps;
    reg=(reg<<1)&mask;
    reg|=parity(t);
    return reg;
}
```

该函数的实现方法是先取出抽头位的数据，然后将寄存器的数据左移一位，并与掩码进行“&”运算，去除最高有效位外的非 0 数据，然后判断抽头数据的奇偶性并添加到寄存器的最低有效位。

clock()函数是根据钟控位的状态来控制寄存器的运行，实现的方法为首先获得三个寄存器的钟控位的状态，然后来确定具体如何运行寄存器，其具体实现代码见程序清单 10-6。

程序清单 10-6

```
void A5_1::clock()
{
    bit maj=majority();
    if(((R1&R1MID)!=0)==maj)
    {
        R1=clockOne(R1,R1MASK,R1TAPS);
    }
    if(((R2&R2MID)!=0)==maj)
    {
        R2=clockOne(R2,R2MASK,R2TAPS);
    }
```

```
        if(((R3&R3MID)!=0)==maj)
        {
            R3=clockOne(R3,R3MASK,R3TAPS);
        }
}
```

clock()函数通过判断移位寄存器钟控位的状态并与三个移位寄存器的钟控位的多数状态进行比较,然后确定是否运行该移位寄存器。

clockAllThree()函数为同时运行 3 个寄存器,具体代码见程序清单 10-7。

程序清单 10-7

```
void A5_1::clockAllThree()
{
    R1=clockOne(R1,R1MASK,R1TAPS);
    R2=clockOne(R2,R2MASK,R2TAPS);
    R3=clockOne(R3,R3MASK,R3TAPS);
}
```

getBit()函数为获得 3 个寄存器的输出,并进行异或运算得到最终输出,函数具体实现代码见程序清单 10-8。

程序清单 10-8

```
bit A5_1::getBit()
{
    return parity(R1&R1OUT)^parity(R2&R2OUT)^parity(R3&R3OUT);
}
```

在完成密钥流生成的前期工作之后,可以通过函数 getKey(byteaToBKeyStream[],bytebToAKeyStream[])来生成密钥流,密钥流分成两部分,分别为 A→B 和 B→A,各 114 位,函数 getKey()的具体实现代码见程序清单 10-9。

程序清单 10-9

```
void A5_1::getKey(byte aToBKeyStream[],byte bToAKeyStream[])
{
    int i;
    for(i=0;i<=113/8;i++)
    {
        aToBKeyStream[i]=bToAKeyStream[i]=0;
    }
    for(i=0;i<114;i++)
    {
        clock();
        aToBKeyStream[i/8]|=getBit()<<(7-(i&7));
    }
    for(i=0;i<114;i++)
    {
        clock();
```

```
16            bToAKeyStream[i/8]|=getBit()<<(7-(i&7));
17        }
18    }
```

代码第 4 行到第 7 行为将 A→B 和 B→A 的密钥流初始化为 0，第 8 行到第 12 行为生成 A→B 的密钥流，第 13 行到第 17 行为生成 B→A 的密钥流。

在密钥流的计算过程中，按照逐位移位并进行或运算的方法实现以“位”为单位的数据转换为 byte 型数据。对应“位”的数据由 getBit()函数将三个移位寄存器的输出进行异或运算获得。

10.3.3 测试

程序的测试部分主要是与已知的正确的生成密钥对比来检查密钥流的生成是否正确，具体代码见程序清单 10-10。

程序清单 10-10

```
void A5_1::run()
{
    byte key[8]={0x12,0x23,0x45,0x67,0x89,0xAB,0xCD,0xEF};
    word frame=0x134;
    byte goodAToB[15]={0x53,0x4E,0xAA,0x58,0x2F,0xE8,0x15,
                        0x1A,0xB6,0xE1,0x85,0x5A,0x72,0x8C,0x00};
    byte goodBToA[15]={0x24,0xFD,0x35,0xA3,0x5D,0x5F,0xB6,
                        0x52,0x6D,0x32,0xF9,0x06,0xDF,0x1A,0xC0};
    byte aToB[15],bToA[15];
    int i;
    setKey(key,frame);
    getKey(aToB,bToA);
    for(i=0;i<15;i++)
    {
        if(aToB[i]!=goodAToB[i])
        {
            cout<<"A to B Failed!"<<endl;
        }
    }
    cout<<endl;
    for(i=0;i<15;i++)
    {
        if(bToA[i]!=goodBToA[i])
        {
            cout<<"B to A Failed!"<<endl;
        }
    }
    cout<<endl;
}
```

代码第 5 行到第 8 行为已知的 A→B 和 B→A 密钥流，第 11 行到第 12 行为生成密钥流，第 13 行到第 27 行为将生成的密钥流与已知的密钥流进行对比来判断生成的密钥流是否正确。

在判断输出的密钥是否正确时，没有采用对密文进行加密和解密的方法来实现，这是因为 A5/1 的加密和解密过程都是采用“异或”来进行的，而 A^B^A＝B，因此采用已知的密钥流来判断过程是否正确更为合适。

10.4 习题与实践题

10.4.1 习题

1. 简要说明序列密码算法的加密和解密的基本原理。

2. 参考图 10-5 的简单的线性反馈移位寄存器，假设输入是 10110，试给出该线性反馈移位寄存器的输出序列。

3. 简要说明 A5/1 线性反馈移位寄存器的工作原理。

4. 简要说明 A5/1 加密算法的基本原理。

10.4.2 实践题

A5/2 算法实现：

A5/2 算法与 A5/1 算法类似，密钥流的生成也是采用线性反馈移位寄存器，但由于其安全性较差，目前只有少数一些国家的通信公司仍在使用。

A5/2 密钥流生成的基本结构见图 10-8。

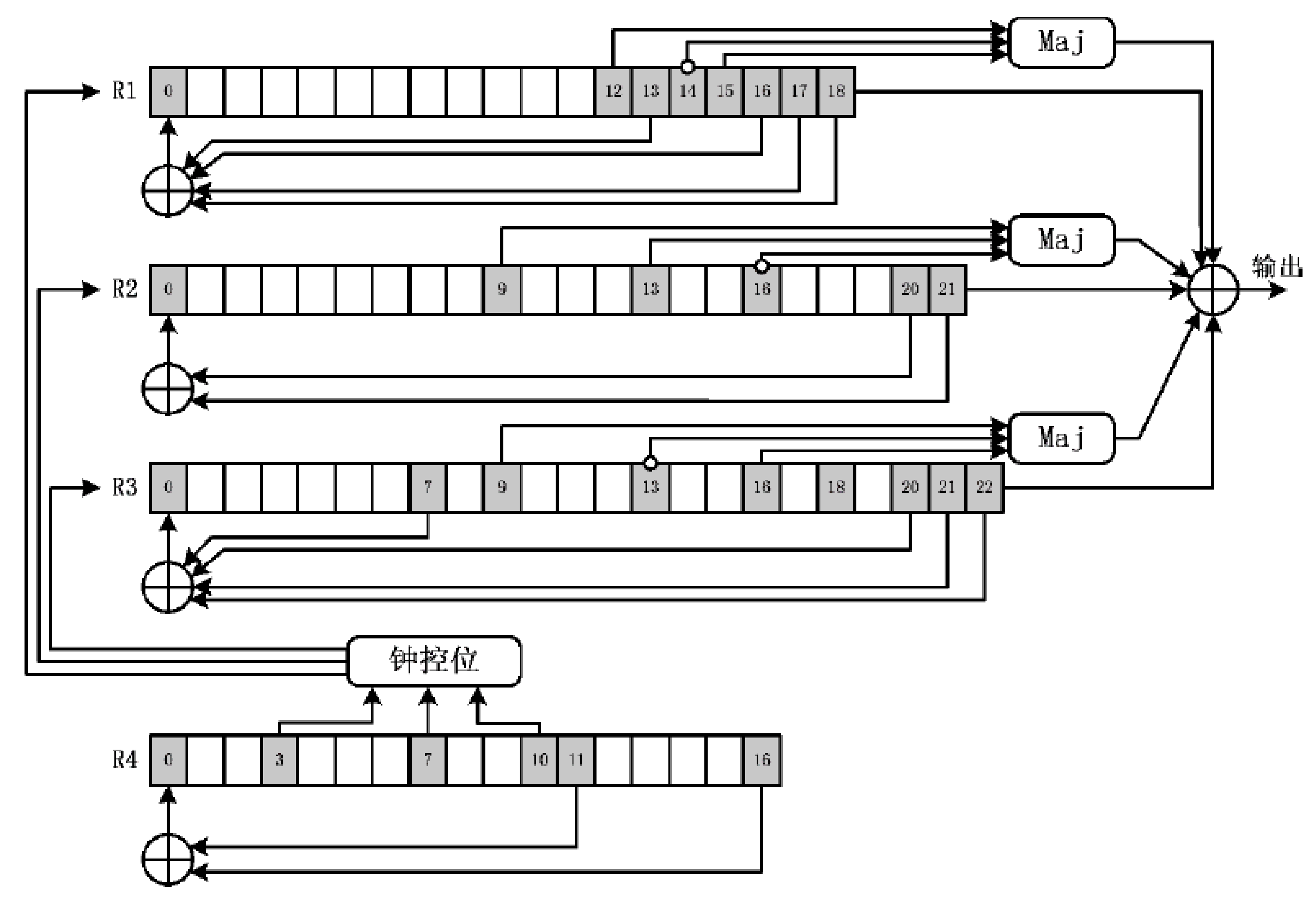


图 10-8 A5/2 密钥流生成的基本结构图

A5/2 算法与 A5/1 算法不同，A5/2 算法使用了 4 个移位寄存器，使用第 4 个寄存器来控制其他 3 个寄存器的运行，各线性反馈移位寄存器的基本参数见表 10-4。

表 10-4 A5/2 寄存器的基本参数

寄存器	长度	特征多项式	钟控位	输入位索引
R_1	19	$x^{19}+x^5+x^2+x+1$	无	13,16,17,18
R_2	22	$x^{22}+x+1$	无	20,21
R_3	23	$x^{23}+x^{15}+x^2+x+1$	无	7,20,21,22
R_4	17	$x^{17}+x^5+1$	R_4[3],R_4[7],R_4[10]	11,16

A5/2 在输出上也与 A5/1 的方法完全不同，输出分为两部分，一部分是通过移位寄存器来输出，即输出 R_1,R_2 和 R_3 的最高有效位，另一部分是通过抽头并确定多数位来输出。将两部分输出的内容进行异或后得到最终的输出。具体抽头的方法见表 10-5。

表 10-5 A5/2 输出方法

寄存器	MSB	多数位	非多数位
R_1	R_1[18]	R_1[12],R_1[15]	R_1[14]
R_2	R_2[21]	R_2[9],R_2[13]	R_2[16]
R_3	R_3[22]	R_3[16],R_3[18]	R_3[13]

A5/1 算法中生成伪随机数的具体步骤如下：

(1) 将所有寄存器置 0。

(2) 寄存器运行 64 个周期，在每个周期 $i(0\leqslant i\leqslant 63)$ 中，第 i 位密钥 $K_c[i]$ 与寄存器的最低有效位(LSB)进行异或运算，并将得到的数据存储于同一寄存器的最低有效位。

(3) 将数据位 R_1[15]、R_2[16]、R_3[18]和 R_4[10]置为 1。

(4) 通过 99 个周期的循环来混淆 K_c 和 F_n，并且不进行输出。钟控方式为：由寄存器 R_4 的钟控单元来计算多数位，即计算 R_4[3],R_4[7]和 R_4[10]的多数位，由多数位来确定寄存器的运行，其具体的运行方式见表 10-6。

表 10-6 寄存器运行方式

R_4[3]	R_4[7]	R_4[10]	多数位	R_1	R_2	R_3
0	0	0	0	clock	clock	clock
1	0	0	0	clock		clock
0	1	0	0	clock	clock	
1	1	0	1		clock	clock
0	0	1	0		clock	clock
1	0	1	1	clock	clock	
0	1	1	1	clock		clock
1	1	1	1	clock	clock	clock

寄存器 R_4 在每个时钟周期内均是最后运行。

(5) 与上一步的运行过程相同，运行 228 个周期，在第 i 次$(0\leqslant i\leqslant 227)$运行过程中将三个寄存器的最高有效位(MSB)输出后进行异或运算，得到伪随机数的第 i 位输出。

程序设计要求： 参考教材中 A5/1 算法，完成 A5/2 算法。

第11章 RC4 算 法

RC4 加密算法是由 Ron Rivest 设计的流加密算法，主要使用软件流加密过程。在安全传输层协议（Transport Layer Security，TLS）、无线加密协议中都采用了 RC4 算法，RC4 算法在应用中提供保密性和数据完整性服务。RC4 算法是密钥长度可变的加密算法，其核心部分的 S 盒长度可为 40 位到 2048 位之间的任意长度，但一般为 256 字节。

11.1 RC4 算法原理

RC4 算法的原理比较简单，首先通过相应算法生成伪随机密钥流，该随机密钥流用于加密和解密过程，加密过程是将明文和密钥流进行“异或”运算，解密的方法和加密的方法相同。

RC4 算法的核心在于密钥流的生成，密钥流生成的第 1 步是生成 S 序列，S 序列可以使用数组表示，序列长度为 256 的 S 序列初始化过程如图 11-1 所示。

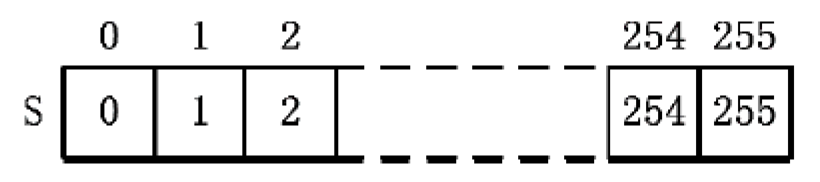


图 11-1　S 序列初始化过程示意图

S 序列的初始化就是将序列索引的值赋给序列当前位置。在 S 序列初始化之后，使用输入的密钥对 S 序列进行置换操作，操作过程如下：

- 计算 j=j+S[i]+key[i mod keylength]
- 交换 S[i]和 S[j]

具体置换过程见图 11-2。

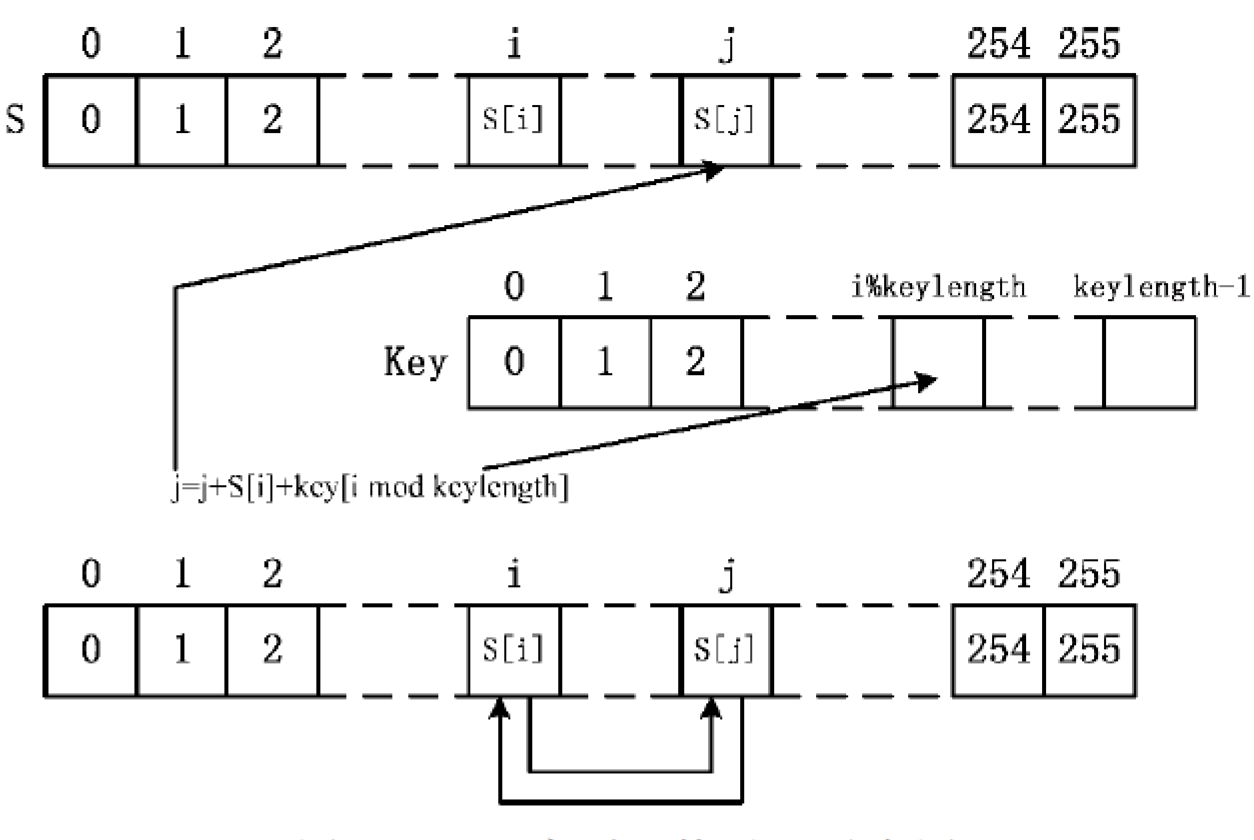


图 11-2　S 序列置换过程示意图

以上两步操作过程可以用伪码表示如下：

```
for i from 0 to 255
    S[i]:=i
endfor
j:=0
for i from 0 to 255
    j:=(j+S[i]+key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

密钥的长度范围通常为 1≤keylength≤256 字节，比较典型的使用范围为 5 个字节到 16 个字节之间，即密钥的长度在 40 位到 128 位之间。

在获得置换后的 S 序列之后，就可以生成 RC4 加密和解密所使用的密钥流，RC4 算法的密钥流长度和所要加密数据的长度相同，具体每个密钥的生成方法如图 11-3 所示。

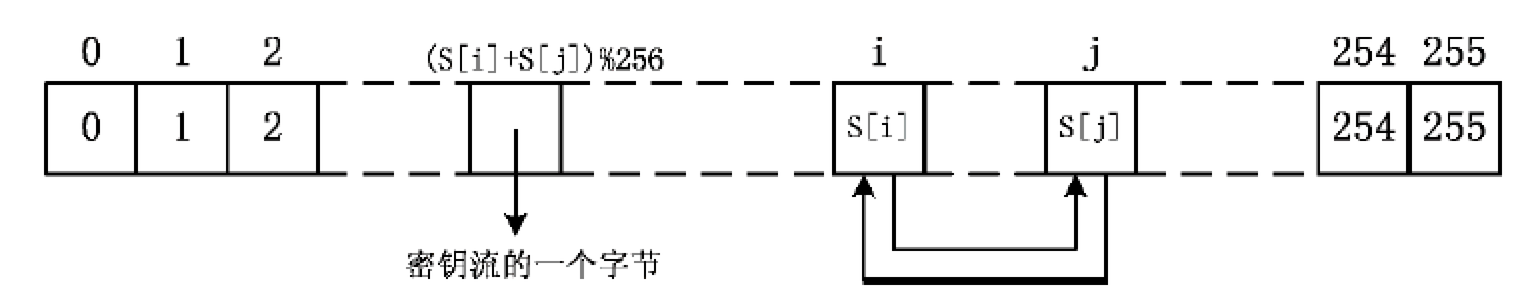


图 11-3 RC4 密钥的生成方法

RC4 密钥的具体生成方法的伪码如下：

```
i:=0
j:=0
while GeneratingOutput
    i:=(i+1) mod 256
    j:=(j+S[i]) mod 256
    swap values of S[i] and S[j]
    K:=S[(S[i]+S[j]) mod 256]
    output K
endwhile
```

在获得密钥流之后，对明文的加密过程非常简单，将相同长度的明文与密钥流进行异或运算就得到密文，具体过程如图 11-4 所示。

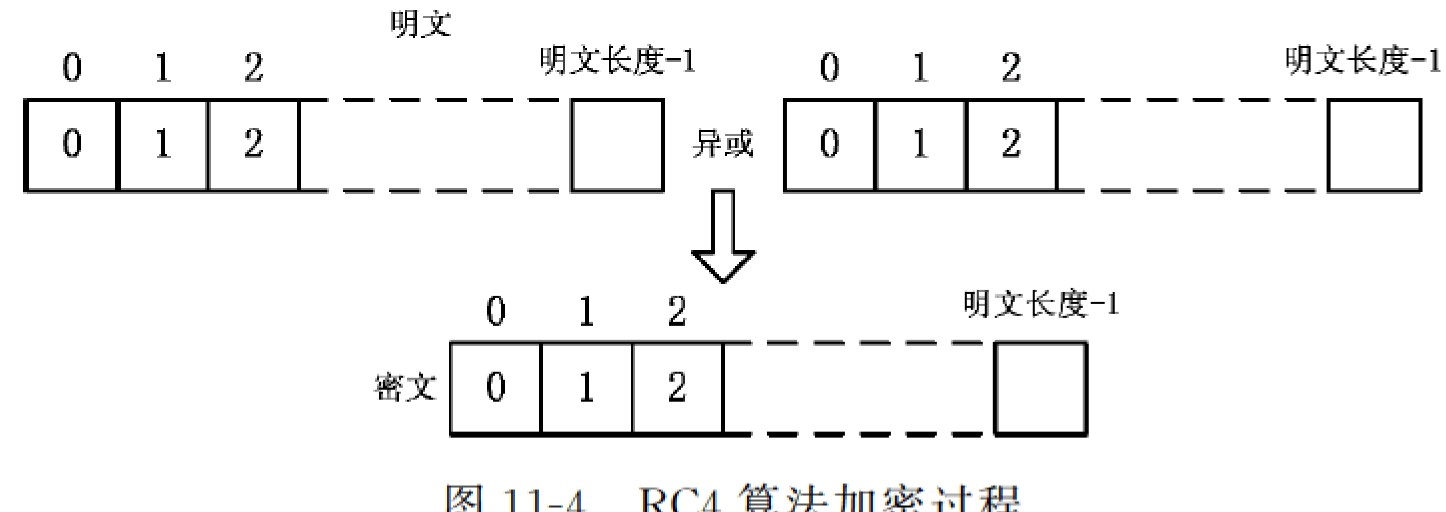


图 11-4 RC4 算法加密过程

RC4 算法的解密过程与 RC4 算法的加密过程相似，具体解密过程如图 11-5 所示。

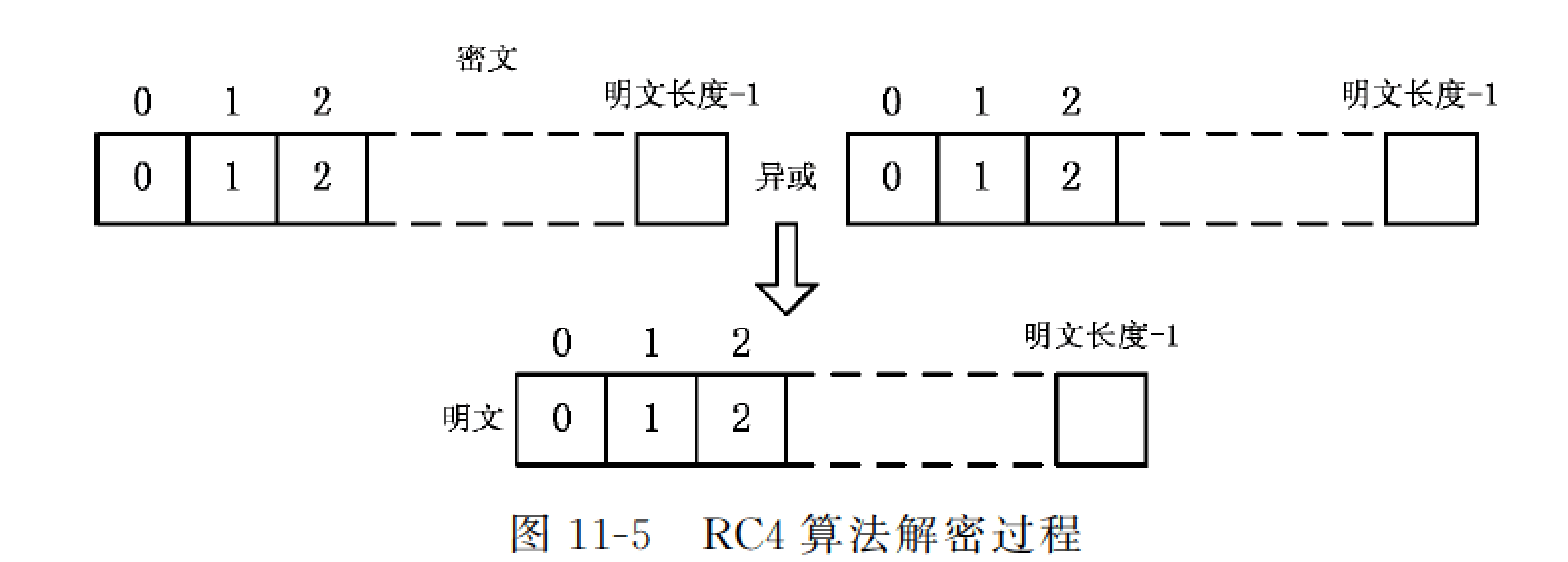


图 11-5 RC4 算法解密过程

11.2 RC4 算法实现

RC4 算法的实现主要分为两部分：一部分是 S 序列的初始化，另一部分是加密或解密。在加密或解密的过程中还需要生成加密密钥或解密密钥，RC4 算法的加密过程和解密过程完全相同。

11.2.1 RC4 算法实现的基本结构

RC4 算法通过 RC_4 类来实现，其基本结构见图 11-6。

RC_4	
- S[256]	: byte
- *key	: byte
- keyLength	: int
- length	: int
- *plainText	: byte
- *cipherText	: byte
- *deCipherText	: byte
+ initS ()	: void
+ setKey (byte *newKey, int newKeyLength)	: void
+ setPlainText (byte *newPlainText, int plainTextLength)	: void
+ swap (byte S[], int i, int j)	: void
+ encryption ()	: void
+ decryption ()	: void
+ showResult ()	: void
+ ~RC_4 ()	

图 11-6 RC_4 类的基本结构

RC_4 类的声明见程序清单 11-1。

程序清单 11-1

```
typedef unsigned char byte;
class RC_4
{
    public:
        void initS();
        void setKey(byte * newKey,int newKeyLength);
        void setPlainText(byte * newPlainText,int plainTextLength);
        void swap(byte S[],int i,int j);
```

```
        void encryption();
        void decryption();
        void showResult();
        ~RC_4();
    private:
        byte S[256];
        byte * key;
        int keyLength;
        int length;
        byte * plainText;
        byte * cipherText;
        byte * deCipherText;
};
```

各成员变量的用途如下：

- S[256]——用于存储 S 序列。
- key——用于存储密钥，由于 RC4 算法中使用的密钥长度是可变的，因此将 key 声明为指针型变量，根据实际密钥的长度来动态分配内存空间。
- keyLength——用于存储密钥的长度。
- length——用来存储明文的长度。
- plainText——用于存储明文，由于 RC4 算法中明文的长度也是可变的，因此将 plainText 声明为指针型变量，根据明文的长度来动态分配内存空间。
- cipherText——用来存储加密后的密文，由于密文的长度是根据明文的长度来确定，因此，同样声明为指针型变量，根据明文的长度来动态分配内存空间。
- deCipherText——用来存储解密后的明文，由于解密后的明文长度也是根据明文的长度来确定，因此，同样声明为指针型变量，根据明文的长度来动态分配内存空间。

各成员函数的作用如下：

- initS()——用于初始化 S 序列。
- setKey()——用于设置密钥，以及获得密钥的长度。
- setPlainText()——用于设置明文，以及获得明文的长度。
- swap()——用于数据交换。
- encryption()——用于对明文加密。
- decryption()——用于对密文解密。
- ～RC_4()——RC_4 类的析构函数，用于释放内存。
- showResult()——用于显示示例中加密和解密的结果。

11.2.2 初始化

RC4 算法实现过程中的初始化包括两部分内容：一部分是获得密钥和密钥长度，一部分是初始化 S 序列。

获得密钥和密钥长度通过函数 setKey()来实现，setKey()函数的代码见程序清单 11-2。

程序清单 11-2

```
void RC_4::setKey(byte * newKey,int newKeyLength)
```

```
02 {
03     int i;
04     keyLength=newKeyLength;
05     key=new byte[keyLength];
06     for(i=0;i<keyLength;i++)
07     {
08         key[i]=newKey[i];
09     }
10 }
```

setKey()函数的参数为密钥和密钥的长度。由于 RC4 加密算法的输入密钥的长度是可变的,函数的参数不仅包含了输入的密钥,还包含了输入密钥的长度。在 setKey()函数中使用了动态数组的方法来处理输入密钥,代码行第 5 行根据输入密钥的长度动态分配密钥空间。

S 序列的初始化通过函数 initS()来实现,initS()函数的详细代码见程序清单 11-3。

程序清单 11-3

```
01 void RC_4::initS()
02 {
03     int i;
04     for(i=0;i<256;i++)
05     {
06         S[i]=i;
07     }
08     int j=0;
09     for(i=0;i<255;i++)
10     {
11         j=(j+S[i]+key[i%keyLength])&0xFF;
12         swap(S,i,j);
13     }
14 }
```

initS()函数实现了两个功能:其一是赋值,代码行第 4 行到第 7 行实现了赋值功能。其二是混乱,代码行第 9 行到第 13 行实现了混乱功能。由于输入密钥的长度是可变的,而 S 序列的长度为 256,因此,需要重复使用输入密钥,在函数中通过 i%keyLength 来处理,当输入密钥使用完之后,再从头开始重复使用。在第 11 行代码中的 &0xFF 的作用是将 j 的范围限制在 0～255 之间,其本质是取 j 的 8 位数据作为有效位。在实现混乱功能中使用了 swap()函数进行数据交换,该函数的详细代码见程序清单 11-4。

程序清单 11-4

```
void RC_4::swap(byte S[],int i,int j)
{
    byte temp=S[i];
    S[i]=S[j];
    S[j]=temp;
}
```

swap()函数是一个简单的数据交换函数，实现的功能为将 S[]数组中的第 i 个元素与第 j 个元素进行交换。

11.2.3 加密和解密

RC4 的加密和解密过程可以通过两种方法来实现，一种方法是生成与明文长度相同的密钥流，然后进行加密和解密。另一种方法是边生成密钥边加密或解密，直到整个加密和解密过程结束。在本 RC4 算法实现中采用的是第二种方法。

在具体加密前需获得要加密的明文，获取明文通过函数 setPlainText()来完成，该函数的详细代码见程序清单 11-5。

程序清单 11-5

```
void RC_4::setPlainText(byte * newPlainText,int plainTextLength)
{
      int i;
      length=plainTextLength;
      plainText=new byte[length];
      for(i=0;i<length;i++)
      {
            plainText[i]=newPlainText[i];
      }
}
```

setPlainText()函数的参数为输入的明文和明文的长度。函数的功能是根据输入明文的内容与长度，将输入的明文赋给类的成员变量 plainText。由于输入的明文长度是可变的，因此该过程也是通过建立动态数组的方法进行处理。

在获得明文之后就可以对明文进行加密，加密过程通过函数 encryption()来完成，该函数的详细代码见程序清单 11-6。

程序清单 11-6

```
void RC_4::encryption()
{
      initS();
      cipherText=new byte[length];
      int i,j,l=0;
      int k=length;
      for(i=0,j=0;k>0;l++,k--)
      {
            i=(i+1)&0xFF;
            j=(j+S[i])&0xFF;
            swap(S,i,j);
            byte K=S[(S[i]+S[j])&0xFF];
            cipherText[l]=K^plainText[l];
      }
}
```

代码行的第 9 行到第 12 行为生成对当前明文加密所需要的密钥，在生成密钥中用到的函数 swap()为程序清单 11-4 中的函数。

解密函数的实现方法与加密函数的实现方法相同，具体代码见程序清单 11-7。

程序清单 11-7

```
void RC_4::decryption()
{
    initS();
    deCipherText=new byte[length];
    int i,j,l=0;
    int k=length;
    for(i=0,j=0;k>0;l++,k--)
    {
        i=(i+1)&0xFF;
        j=(j+S[i])&0xFF;
        swap(S,i,j);
        byte K=S[(S[i]+S[j])&0xFF];
        deCipherText[l]=K^cipherText[l];
    }
}
```

加密函数和解密函数也可以通过一个函数来实现，可以将函数声明如下：

```
void encryption(byte * in,byte * out);
```

函数具体实现的方法与 encryption()相同。

11.2.4 RC4 算法测试

RC4 算法实现过程的测试可以通过主函数来进行，通过输入明文、密钥对加密和解密过程进行测试，由于加密和解密过程均采用“异或”来实现，因此，在测试过程中采用已知的测试数据来进行，具体的测试过程见程序清单 11-8。

程序清单 11-8

```
int main()
{
    RC_4 rc4;
    byte plainText[14]={'A','t','t','a','c','k',' ','a','t',' ','d','a','w','n'};
    byte key[6]={'S','e','c','r','e','t'};
    rc4.setPlainText(plainText,14);
    rc4.setKey(key,6);
    rc4.encryption();
    rc4.decryption();
    rc4.showResult();
    return 0;
}
```

测试的结果通过函数 showResult()来显示，showResult()函数的具体代码见程序清单 11-9。

程序清单 11-9

```
void RC_4::showResult()
{
      int i;
      cout<<endl;
      cout<<"PlainText=";
      for(i=0;i<length;i++)
      {
            cout<<plainText[i];
      }
      cout<<endl;
      cout<<"Key=";
      for(i=0;i<keyLength;i++)
      {
            cout<<key[i];
      }
      cout<<endl;
      cout<<"Cipher Text=";
      for(i=0;i<length;i++)
      {
            cout<<hex<<int(cipherText[i]);
      }
      cout<<endl;
      cout<<"DeCipher Text=";
      for(i=0;i<length;i++)
      {
            cout<<deCipherText[i];
      }
      cout<<endl;
}
```

测试运行得到的结果如下：

```
PlainText=Attack at dawn
Key=Secret
Cipher Text=45a01f645fc35b383552544b9b67
DeCipher Text=Attack at dawn
```

加密的结果是以十六进制的形式输出。

在程序的运行过程中采用了动态分配内存的方法来处理相关数据，因此在对象使用完毕后需释放相关内存，释放内存的任务由析构函数来实现，析构函数所需要释放的内存包括密钥、明文、加密后的密文和解密后的明文，具体实现代码见程序清单 11-10。

程序清单 11-10

```
RC_4::~ RC_4()
{
    delete []key;
    delete []plainText;
    delete []cipherText;
    delete []deCipherText;
}
```

析构函数将所有动态分配的内存释放。

11.3 习题与实践题

11.3.1 习题

1. 简要说明 RC4 算法的 S 序列初始化的基本原理。
2. 简要说明 RC4 算法的加密和解密密钥流生成的基本原理。
3. 简要说明 RC4 算法的加密和解密的基本过程。

11.3.2 实践题

参考 11.2 节中 RC4 算法的实现方法,编程实现 RC4 加密和解密算法。要求:待加密的数据从文件读取,加密后的数据保存到文件,解密后的数据也保存到相应文件。程序可以比较加密和解密的过程是否正确,即比较待加密的消息和解密后的消息是否相同。

第 12 章

RC5 算 法

RC5 算法是由 Ronald L. Rivest 设计的对称分组加密算法，是一种可变加密轮数、可变分组长度、可变加密密钥长度的算法。RC5 算法既适合于硬件加密，也适合于软件加密。其加密方法和解密方法都特别简单，同时，还具有较高的加密速度。在无线传输层安全(Wireless Transport Level Security，WTLS)规范中使用 RC5 算法作为无线客户端与服务器之间的加密算法。

12.1 RC5 算法原理

RC5 加密算法具有三个可变的特征：

- 加密轮数可变——加密的轮数可以在 0～255 轮之间，通常使用的加密轮数为 12 轮。
- 分组长度可变——分组可选择 32 位、64 位或 128 位的分组长度，通常使用 64 位作为分组长度。
- 密钥长度可变——加密和解密的密钥长度可以在 0 位到 2040 位之间，通常使用的密钥的长度为 128 位。

RC5 可以用类似于 RC5-$w/r/b$ 的方法来表示具体的算法，w 表示 word 的位数，r 表示加密的轮数，b 表示密钥的长度(以 8 位字节为单位)，见表 12-1。

表 12-1 RC5 算法表示说明

参数	定义	取值范围
w	以 bit 表示的 word 的尺寸	16，32，64
r	加密与解密轮数	$0\leqslant r\leqslant 255$
b	密钥的字节长度	$0\leqslant b\leqslant 255$

例如，RC5-32/12/16，其中 32 表示处理比特的字长为 32 位，由于 RC5 算法处理的数据是 2 个字长，因此分组长度为 64 比特，12 表示加密和解密的轮次，16 表示密钥长度为 16 字节，每个字节 8 位，共 128 位。Rivest 建议的加密和解密的版本为 RC5-32/12/16。

12.1.1 RC5 加密和解密的基本原理

RC5 加密和解密算法的基本原理如图 12-1 所示。

图 12-1 是 RC5 算法中的一轮运算过程，图中的 A，B 是输入的明文分组(长度为 $2w$ 比特)分为长度为 w 比特的字(word)，S 为子密钥。RC5 算法中使用了 3 种基本的运算：

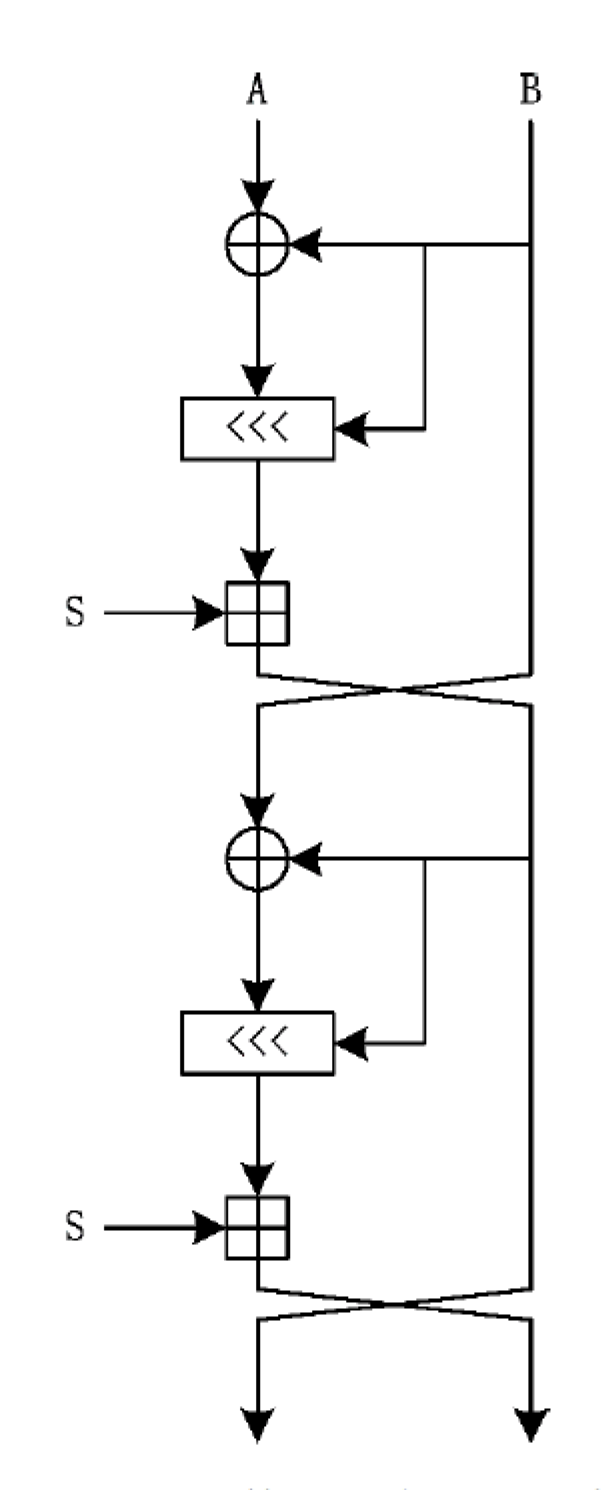


图 12-1 RC5 算法基本原理示意图

(1) ⊞——模 2 加运算。

(2) ⊕——异或运算。

(3) ——循环左移。

RC5 加密算法用伪码可以描述为

```
A:=A+S[0]
B:=B+S[1]
for i from 1 ro r
    A:=((A⊕B)<<<B)+S[2*i]
    B:=((B⊕A)<<<A)+S[2*i+1]
endfor
```

伪码中的 r 表示加密的轮数。

RC5 解密算法用伪码可以描述为

```
for i from 1 to r
    B:=((B-S[2*i+1])>>>A)⊕A
    A:=((A-S[2*i])>>>B)⊕B
endfor
B:=B-S[1]
A:=A-S[0]
```

在加密和解密过程中均用到密钥 S[i],密钥 S[i]通过密钥扩展获得。

12.1.2 RC5 密钥生成

RC5 算法的密钥扩展是通过用户输入的密钥 K 填充密钥序列 S 来进行,在密钥初始化

过程中使用了两个称为“魔数常量”的特殊常量，分别为 P_w 和 Q_w，并在运算过程中使用了字长 w 来进行运算，P_w 和 Q_w 的计算方法如下：

$$P_w = \mathrm{odd}((e-2)2^w)$$
$$Q_w = \mathrm{odd}((\phi-1)2^w) \tag{12-1}$$

其中：

$e=2.718281828459\cdots$ （自然对数底）

$\phi=1.618033988749\cdots$ （黄金分割）

$\mathrm{odd}(x)$是指 x 向上取整并最接近 x 的奇数

w 的取值一般为 16，32 或 64，对于不同的 w 计算获得的 P_w 和 Q_w 为

$P_{16} = (1011011111100001)_2 = \mathrm{0xB7E1}$

$Q_{16} = (1001111000110111)_2 = \mathrm{0x9E37}$

$P_{32} = (10110111111000010101000101100011)_2 = \mathrm{0xB7E15163}$

$Q_{32} = (10011110001101110111100110111001)_2 = \mathrm{0x9E3779B9}$

$P_{64} = (1011011111100001010100010110001010001010111011010010101001101011)_2$
$= \mathrm{0xB7E151628AED2A6B}$

$Q_{64} = (1001111000110111011110011011100101111111010010100111110000010101)_2$
$= \mathrm{0x9E3779B97F4A7C15}$

RC5 密钥生成的过程可以分为以下 3 个步骤：

（1）将用户以字节为单位的密钥数组转换为以字为单位的密钥数组

由于在加密过程中的密钥 S 使用的密钥是以字(word)为单位，而输入的密钥是以字节为单位，因此还需要将输入的以字节为单位的密钥转换为字。

假设有以字节为单位的密钥数组 K[]，数组长度为 b，并有以字为单位的数组 L[]，数组长度为 c，c 与 b 的关系为

$$c = \lceil b \times 8/w \rceil \tag{12-2}$$

式(12-2)表示 L[]数组的大小正好能容纳用户输入密钥数组的大小，如果用户输入的密钥的 bit 长度不是字长的整数倍，那么在填充数组 L[]的最后一个字时，在使用完所有 K[]数组数据之后使用“0”来填充后续字节。

（2）初始化 S 序列

在将输入的密钥转换为以字为单位的数组之后的第二步为初始化 S 序列，初始化 S 序列使用了 P_w 和 Q_w，具体初始化过程的伪码如下：

```
S[0]:=Pw
for i from 1 to t
    S[i]:=S[i-1]+Qw
endfor
```

其中 $t=2*r+2$。

由于 S 序列的每个元素的长度为字长，因此在计算过程中还需要将获得的结果进行模 2^w 运算。

(3) 密钥混淆

密钥混淆过程是将 S 序列与用户输入的密钥进行混淆操作，在这步操作过程中使用在(1)中获得的 L[]数组来对 S[]序列进行混淆操作，具体混淆过程的伪码如下：

```
i:=0
j:=0
A:=0
B:=0
do 3×max(t,c) times
    A:=S[i]=(S[i]+A+B)<<<3
    B:=L[j]=(L[j]+A+B)<<<(A+B)
    i:=(i+1) mod (t)
    j:=(j+1) mod (c)
enddo
```

12.2 RC5 算法实现

RC5 算法的实现包含加密和解密以及密钥生成两大部分，密钥的生成又由三部分的操作完成：用户密钥的转换、S 序列的初始化和 S 序列的混淆。与 RC4 算法相比较，RC4 的加密过程与解密过程完全一样，而 RC5 算法中的加密和解密算法略有不同。

在本算法实现中使用的 RC5 结构为 RC5-32/12/16，即字长为 32 位、加密轮数为 12 轮、用户密钥长度为 16 字节(128 位)。

12.2.1 RC5 算法实现的基本结构

RC5 算法实现的主要功能通过 RC_5 类来完成，RC_5 类的基本结构如图 12-2 所示。

RC_5	
- L[c]	: word
- S[t]	: word
- plainText[2]	: word
- cipherText	: word
- deCipherText	: word
+ setKey (unsigned char userKey[])	: void
+ setPlainText (word pText[])	: void
+ rotL (word x, word y)	: word
+ rotR (word x, word y)	: word
+ encryption (word in[], word out[])	: void
+ decryption (word in[], word out[])	: void
+ test ()	: void

图 12-2 RC_5 类的基本结构

RC_5 类的声明见程序清单 12-1。

程序清单 12-1

```
typedef unsigned int word;
const int w=32;
const int r=12;
const int b=16;
const int c=4;
const int t=26;
const word P=0xB7E15163;
const word Q=0x9E3779B9;
class RC_5
{
    public:
        void setKey(unsigned char userKey[]);
        void setPlainText(word pText[]);
        word rotL(word x,word y);
```

```
        word rotR(word x,word y);
        void encryption(word in[],word out[]);
        void decryption(word in[],word out[]);
        void test();
    private:
        word L[c];
        word S[t];
        word plainText[2];
        word cipherText[2];
        word deCipherText[2];
};
```

在程序清单 12-1 中，word 数据类型的定义是为了方便 32 位数据的处理，而常量 w 表示处理数据的位数、r 表示加密的轮数、b 表示输入密钥以字节为单位的长度、c 是输入以字节为单位的密钥转换为以 word 为单位的长度、t 为 2 * r+2。

程序清单 12-1 中各成员变量的用途如下：

- L[c]——转换为 word 格式的用户输入密钥。
- S[t]——密钥序列。
- plainText[2]——输入的明文，每次处理 2 个 word 的数据。
- cipherText——存储将明文加密后获得的密文。
- deCipherText——存储将密文解密后得到的明文。

程序清单 12-1 中各成员函数的作用如下：

- setKey()——通过用户输入的密钥计算得到加密和解密的密钥。
- setPlainText()——获得用户输入的明文。
- rotL()——计算密钥、加密和解密过程中使用的辅助函数，实现循环移位的功能。
- rotR()——计算密钥、加密和解密过程中使用的辅助函数，实现循环移位的功能。
- encryption()——加密函数，用于对明文的加密。
- decryption()——解密函数，用于对密文的解密。
- test()——测试函数，用于测试加密和解密过程。

12.2.2 密钥生成

RC5 算法使用的密钥通过函数 setKey()来完成，函数的参数是用户输入的密钥，数据类型为 unsigned char。函数共实现三个功能：将输入的密钥转换为 word 型数据、初始化 S 序列和密钥混淆。函数具体实现代码见程序清单 12-2。

程序清单 12-2

```
void RC_5::setKey(unsigned char userKey[])
{
    int i,j,k;
    int u=w/8;
    word A,B;
    for(i=b-1,L[c-1]=0;i!=-1;i--)
```

```
07      {
08          L[i/u]= (L[i/u]<< 8)+ userKey[i];
09      }
10      for (S[0]= P,i= 1;i< t;i+ + )
11      {
12          S[i]= S[i- 1]+ Q;
13      }
14      for (A= B= i= j= k= 0;k< 3* t;k+ + ,i= (i+ 1)%t,j= (j+ 1)%c)
15      {
16          A= S[i]= rotL(S[i]+ (A+ B),3);
17          B= L[j]= rotL(L[j]+ (A+ B),(A+ B));
18      }
19  }
```

程序清单 12-2 中的第 6 行到第 9 代码的作用是将用户输入的字节型密钥转换为 word 型密钥，实现的原理见图 12-3。

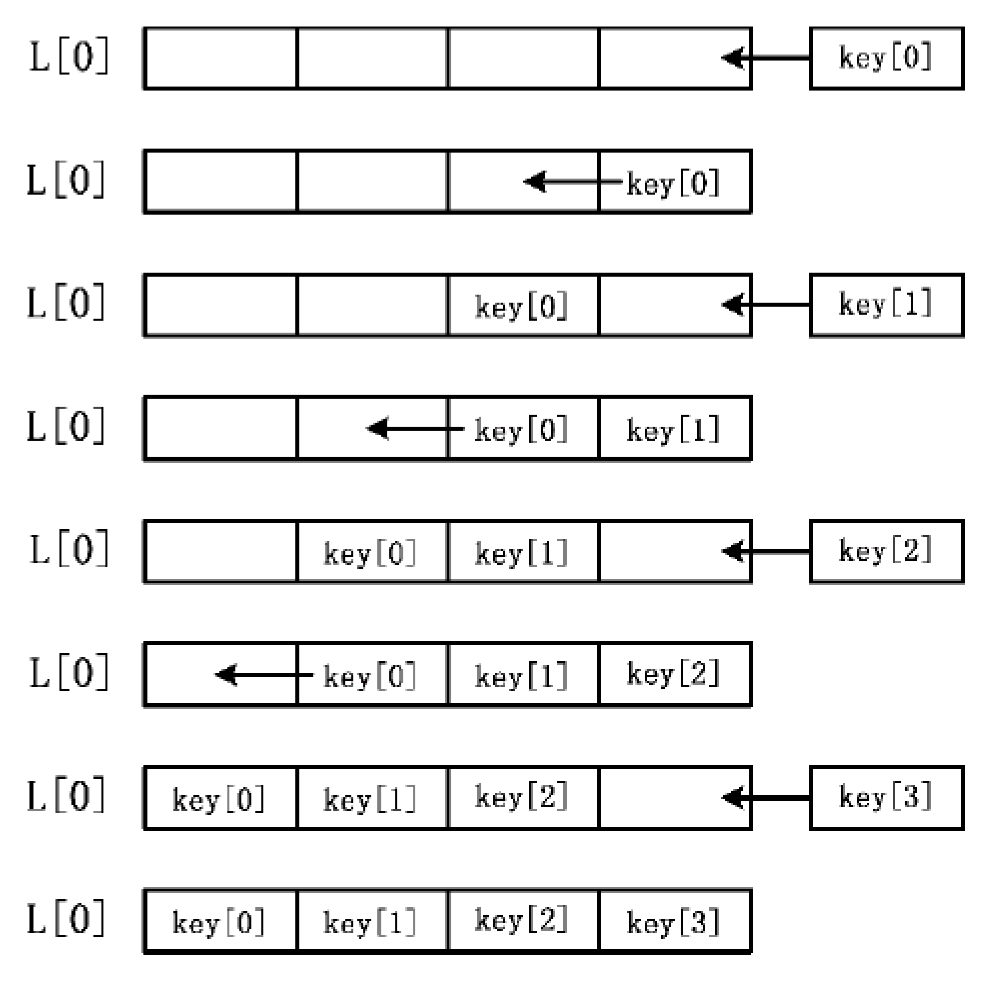


图 12-3　字节型密钥转换为 word 型密钥示意图

图 12-3 显示的是第 1 个 word 型密钥的转换过程，其余密钥的转换方法相同。

代码行的第 10 行到第 13 行为 S 序列初始化，初始化过程通过使用常量 P 和 Q 来完成。代码行的第 14 行到第 18 行为对密钥进行混淆操作，在这过程中使用了 rotL()函数进行了相关的循环移位操作，rotL()函数的参数为两个 word 型参数，rotL()函数的实现过程见程序清单 12-3。

程序清单 12-3

```
word RC_5::rotL(word x,word y)
{
    return (((x)<< (y&(w- 1)))|((x)>> (w- (y&(w- 1)))));
}
```

rotL()函数的参数为待移位的数据 x 和循环移位的多少 y，实现的功能为循环左移。在移位运算过程中使用 y&(w－1)，w 为 32，w－1 为 31，w－1 用二进制数据可以表示为

$(11111)_2$。由于 y 为 word32 型数据，因此，在进行 y&(w－1)之后的最大值为 31，正好用于 32 位数据的循环移位。

12.2.3 加密和解密过程的实现

RC5 的加密和解密过程通过函数 encryption()和 decryption()来实现，每次加密和解密过程都是针对 2 个 word 型数据进行，即每次处理 64 位数据。加密函数 encryption()的具体实现代码见程序清单 12-4。

程序清单 12-4

```
void RC_5::encryption(word in[],word out[])
{
    int i;
    word A=in[0]+S[0];
    word B=in[1]+S[1];
    for(i=1;i<=r;i++)
    {
        A=rotL(A^B,B)+S[2*i];
        B=rotL(B^A,A)+S[2*i+1];
    }
    out[0]=A;
    out[1]=B;
}
```

encryption()函数的参数为 word 型的输入（明文）和 word 型的输出（密文），并使用数组来处理输入和输出。由于每次处理 2 个 word 型数据，因此数组的实际大小为 2。

在加密过程中同样用到了 rotL()函数，rotL()函数见程序清单 12-3。

解密函数 decryption()的具体实现代码见程序清单 12-5。

程序清单 12-5

```
void RC_5::decryption(word in[],word out[])
{
    int i;
    word A=in[0];
    word B=in[1];
    for(i=r;i>0;i--)
    {
        B=rotR(B-S[2*i+1],A)^A;
        A=rotR(A-S[2*i],B)^B;
    }
    out[0]=A-S[0];
    out[1]=B-S[1];
}
```

decryption()函数的参数为 word 型的输入（密文）和 word 型的输出（解密后的明文），也使用大小为 2 的数组。在解密过程中使用了函数 rotR()来处理相关的循环移位操作，

rotR()函数的具体代码见程序清单 12-6。

程序清单 12-6

```
word RC_5::rotR(word x,word y)
{
      return (((x)>>(y&(w-1)))|((x)<<(w-(y&(w-1)))));
}
```

rotR()函数实现的功能是循环右移,实现过程的基本原理与循环左移函数 rotL()相同。

12.2.4 RC5 算法测试

RC5 算法的测试过程通过主函数 main()以及测试函数 test()来进行。通过主函数来设置用户密钥和明文,通过 test()函数来测试加密和解密过程。设置明文的函数为 setPlainText(),函数的具体代码见程序清单 12-7。

程序清单 12-7

```
void RC_5::setPlainText(word pText[])
{
      plainText[0]=pText[0];
      plainText[1]=pText[1];
}
```

setPlainText()的参数为 word 型数组,数组大小为 2。

测试函数 test()的具体代码见程序清单 12-8。

程序清单 12-8

```
void RC_5::test()
{
      cout<<"The plainText:"<<endl;
      cout<<hex<<word(plainText[0])<<" ";
      cout<<hex<<word(plainText[1])<<endl;
      encryption(plainText,cipherText);
      cout<<"The cipherText:"<<endl;
      cout<<hex<<word(cipherText[0])<<" ";
      cout<<hex<<word(cipherText[1])<<endl;
      cout<<"The deCipherText:"<<endl;
      decryption(cipherText,deCipherText);
      cout<<hex<<word(deCipherText[0])<<" ";
      cout<<hex<<word(deCipherText[1])<<endl;
}
```

测试过程通过主函数来驱动,主函数的具体代码见程序清单 12-9。

程序清单 12-9

```
int main()
```

```
{
    RC_5 rc5;
    unsigned char key[b];
    int i;
    for(i=0;i<b;i++)
    {
        key[i]=0;
    }
    rc5.setKey(key);
    word pText[2];
    pText[0]=0x9ABCDEF0;
    pText[1]=0x12345678;
    rc5.setPlainText(pText);
    rc5.test();
    return 0;
}
```

测试过程首先输入密钥，然后计算 RC5 算法所用的密钥，再设置明文，通过调用 test()函数来进行加密和解密过程测试。测试结果如下：

```
The plainText:
9abcdef0 12345678
The cipherText:
5c65922 2865cf3
The deCipherText:
9abcdef0 12345678
```

在对明文加密后再进行解密得到解密后的明文与原先输入的明文一致。

12.3 习题与实践题

12.3.1 习题

1. RC5 算法中用到哪些基本运算？给出算例说明这些运算的基本方法。
2. 简要说明 RC5 加密算法的基本原理。
3. 简要说明 RC5 加密算法密钥的生成方法和基本过程。

12.3.2 实践题

1. RC5 算法中的核心部分之一是密钥的生成，试编写一程序，程序的功能是生成 RC5 加密算法的密钥，并将密钥保存到相关文件。

2. 编写 RC5 加密算法，要求：待加密的消息是从文件读取，加密后的消息保存到文件，并可以对解密结果进行检验。

第 13 章

RC6 算法

RC6 算法也是由 Ron Rivest 设计的一种加密算法，RC6 算法是以 RC5 算法为基础，曾是高级加密标准的候选算法。RC6 算法的基本思想是广泛采用数据移位来增强抵抗攻击的能力，同时采用了同时处理 4 个明文分组的结构，并在运算中增加了扩散的行为，使得即使使用较少的加密轮数也具有较高的安全性。

13.1 RC6 算法原理

RC6 算法与 RC5 算法相似，也是由三个参数决定的算法簇。决定具体算法的参数分别为 w(word 字长)、r(加密轮数)和 b(以 word 为单位的密钥长度)。其中 b 也可以使用字节长度或位长度，转换为字长度只需要做适当的换算即可。因此 RC6 算法通常也可以表示为：RC6-w/r/b，例如 RC6-32/20/32 表示该 RC6 算法为处理的字长为 32 位、加密的轮数为 20、密钥的长度为 32 字节。

13.1.1 RC6 算法的加密和解密

RC6 算法的加密和解密过程的基本原理如图 13-1 所示。

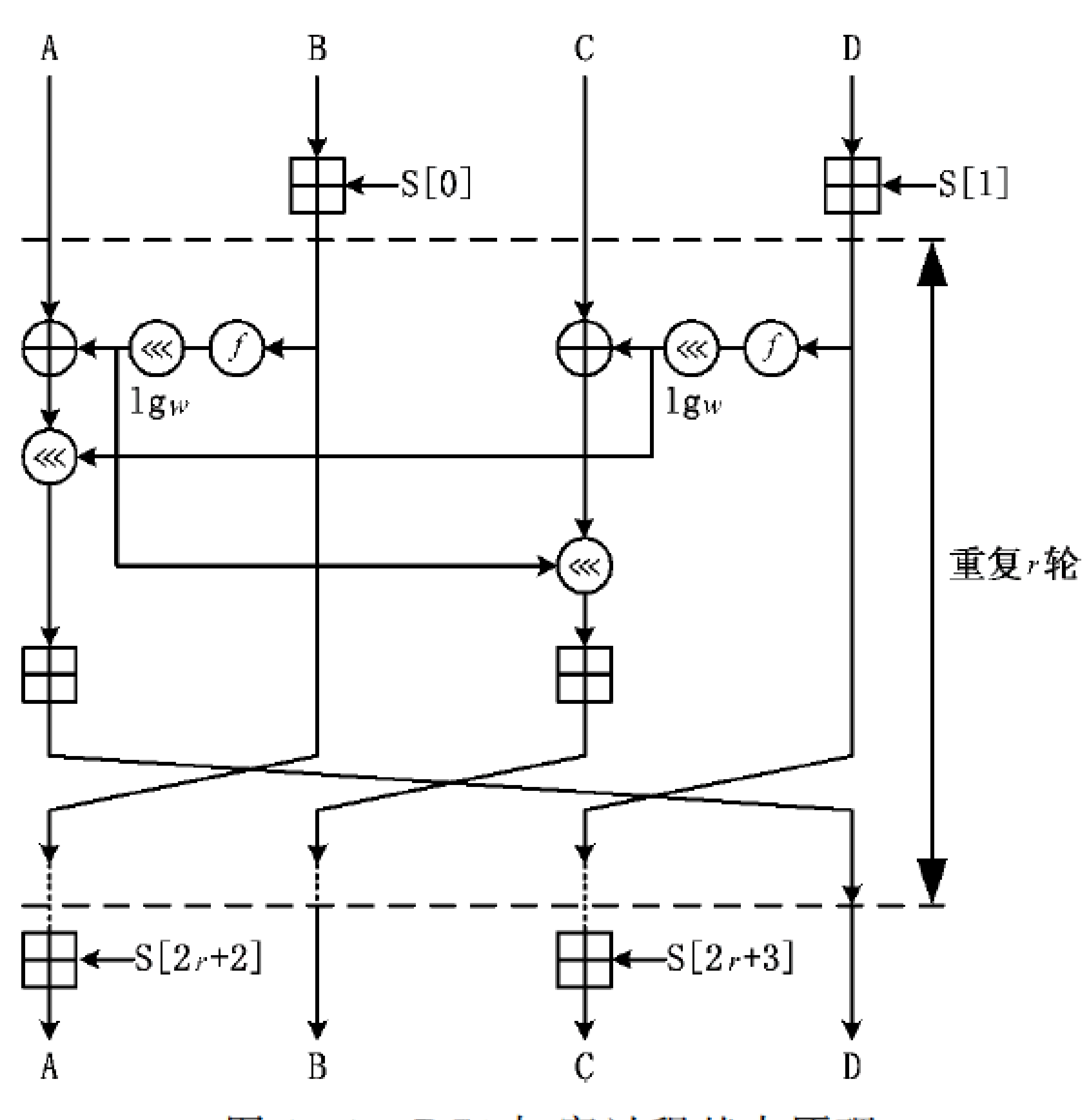


图 13-1　RC6 加密过程基本原理

在 RC6 算法中共用到以下运算：

a+b——整数加法，并将结果模 2^w。

a−b——整数减法，并将结果模 2^w。

a ⊕ b——异或运算。

a×b——整数乘法，并将结果模 2^w。

a　b——循环左移。

a　b——循环右移。

假设明文输入为 A，B，C 和 D，加密轮数为 r 轮，w 位的轮密钥为 S[0，…，$2r+3$]，加密后的输出也存储到 A，B，C 和 D，那么加密过程用伪码可以表示为

```
B:=B+S[0]
D:=D+S[1]
for i from 1 to r
     t:=(B×(2B+1))<<<lgw
     u:=(D×(2D+1))<<<lgw
     A:=((A⊕t)<<<u)+S[2i]
     C:=((C⊕u)<<<t)+S[2i+1]
     (A,B,C,D):=(B,C,D,A)
endfor
A:=A+S[2r+2]
C:=C+S[2r+3]
```

RC6 算法的解密过程与加密过程类似，假设密文输入为：A，B，C 和 D，解密轮数为 r 轮，w 位的轮密钥为 S[0，…，$2r+3$]，解密后的输出也存储到 A，B，C 和 D，那么解密过程用伪码可以表示为

```
C:=C-S[2r+3]
A:=A-S[2r+2]
for i from r to 1
     (A,B,C,D):=(D,A,B,C)
     u:=(D×(2D+1))<<<lgw
     t:=(B×(2B+1))<<<lgw
     C:=((C-S[2i+1])>>>t)⊕u
     A:=((A-S[2i])>>>u)⊕t
endfor
D:=D-S[1]
B:=B-S[0]
```

13.1.2 RC6 算法的密钥生成

RC6 的密钥生成算法与 RC5 的密钥生成算法基本相同，也分成三步完成：首先将用户输入的密钥转换为 word 型（处理单个明文输入的长度）密钥，然后初始化密钥序列，最后对密钥进行混淆。在密钥生成过程中也使用了 P_w 和 Q_w，对于处理 32 位的 word 型数据，P_w 和 Q_w 的值为

```
P32=0xB7E15163
Q32=0x9E3779B9
```

P_w 和 Q_w 的值与 RC5 算法中的 P_w 和 Q_w 的值计算方法和结果都相同。

RC6 密钥的生成与 RC5 密钥的生成的不同地方是密钥的长度不同，假设用户密钥通过转换为 word 型密钥序列为 L[0，…，$c-1$]，w 位密钥序列为 S[0，…，$2r+3$]，那么，RC6 的密钥生成过程可以用伪码描述为

```
S[0]:=Pw
for i from 1 to 2r+3
    S[i]:=S[i-1]+Qw
endfor
A=B=i=j=0
v:=3×max(c,2r+4)
for s from 1 to v
    A:=S[i]=(S[i]+A+B)<<<3
    B:=L[j]=(L[j]+A+B)<<<(A+B)
    i:=(i+1)mod(2r+4)
    j:=(j+1)mod c
endfor
```

在使用用户输入密钥填充 word 型密钥序列的过程中，若用户输入密钥不够则使用 0 字节进行填充。

13.2 RC6 算法实现

RC6 算法实现的主要功能通过 RC_6 类来完成，RC_6 类的主要功能包括生成密钥、加密和解密，加密和解密过程分别通过加密和解密的辅助函数来完成加密和解密的具体过程。RC6 算法的加密过程中每次处理 4 个 word 型数据，在本示例中加密的数据长度为 32 位，每次加密 4 个 word 型数据，共 128 位，总加密或解密轮数为 20 轮，用户输入密钥的长度为 16 字节，共 128 位，生成的密钥序列共 44 个 word 型密钥。

13.2.1 RC6 算法实现的基本结构

RC_6 类的基本结构见图 13-2。

RC_6 类的完整声明见程序清单 13-1。

程序清单 13-1

```
typedef unsigned int word;
const int w=32;
const int r=20;
const int b=16;
const int c=4;
const int t=44;
```

RC_6	
- L[c]	: word
- S[t]	: word
- plainText[4]	: word
- cipherText[4]	: word
- deCipherText[4]	: word
+ setKey (unsigned char userKey[])	: void
+ setPlainText (word pText[])	: void
+ rotL (word x, word y)	: word
+ rotR (word x, word y)	: word
+ encryption (word in[], word out[])	: void
+ encHelper (int i, word &A, word &B, word &C, word &D)	: void
+ decryption (word in[], word out[])	: void
+ decHelper (int i, word &A, word &B, word &C, word &D)	: void
+ test ()	: void

图 13-2 RC_6 类的基本结构

```
const word P= 0xB7E15163;
const word Q= 0x9E3779B9;
class RC_6
{
    public:
        void setKey(unsigned char userKey[]);
        void setPlainText(word pText[]);
        word rotL(word x,word y);
        word rotR(word x,word y);
        void encryption(word in[],word out[]);
        void encHelper(int i,word &A,word &B,word &C,word &D);
        void decryption(word in[],word out[]);
        void decHelper(int i,word &A,word &B,word &C,word &D);
        void test();
    private:
        word L[c];
        word S[t];
        word plainText[4];
        word cipherText[4];
        word deCipherText[4];
};
```

在程序示例中，加密的明文是 4 个 32 位的数据为一分组，因此定义一个新的数据类型 word 以方便后续处理。定义常量 w 用于处理 32 位数据，常量 r 为加密轮数，常量 b 为输入密钥的字节数，常量 c 为将字节型密钥转换为 word 型密钥后的数量，常量 t=2 * r+4，常量 P 和 Q 是用于初始化 S 序列用的常量，与 RC5 中所用的常量的含义相同。

程序清单 13-1 中个成员变量的作用如下：

- L[c]——转换为 word 格式的用户输入密钥。
- S[t]——密钥序列。
- plainText[4]——输入的明文，每次处理 4 个 word 的数据。

- cipherText[4]——存储将明文加密后获得的密文。
- deCipherText[4]——存储将密文解密后得到的明文。

程序清单 13-1 中各成员函数的作用如下：

- setKey()——获得用户输入的密钥，并根据用户输入的密钥计算获得加密与解密用的密钥序列。
- setPlainText()——获取用户输入的明文。
- rotL()——计算密钥、加密和解密过程中使用的辅助函数，实现循环移位的功能。
- rotR()——计算密钥、加密和解密过程中使用的辅助函数，实现循环移位的功能。
- encryption()——加密函数，用于对明文的加密。
- encHelper()——加密辅助函数，用于实现一轮的加密。
- decryption()——解密函数，用于对密文的解密。
- decHelper()——解密辅助函数，用于实现一轮的解密。
- test()——测试函数，用于测试加密和解密过程。

13.2.2 密钥生成

RC_6 类中，密钥生成通过函数 setKey()，函数的参数为 unsigned char 型数组，用于接收用户输入的密钥，setKey()函数共实现三个基本功能：将用户输入的密钥转换为 word 型密钥、初始化密钥序列 S、混淆密钥序列 S。

setKey()函数的详细代码见程序清单 13-2。

程序清单 13-2

```
void RC_6::setKey(unsigned char userKey[])
{
    int i,j,k;
    int u=w/8;
    word A,B;
    for(i=b-1,L[c-1]=0;i!=-1;i--)
    {
        L[i/u]=(L[i/u]<<8)+userKey[i];
    }
    for(S[0]=P,i=1;i<t;i++)
    {
        S[i]=S[i-1]+Q;
    }
    for(A=B=i=j=k=0;k<3*t;k++,i=(i+1)%t,j=(j+1)%c)
    {
        A=S[i]=rotL(S[i]+(A+B),3);
        B=L[j]=rotL(L[j]+(A+B),(A+B));
    }
}
```

RC_6 中生成密钥的算法原理和过程与 RC_5 中生成密钥的算法和原理完全相同，只是生成密钥的数量不同，在 RC_6 算法中生成密钥的数量为 44 个 word 型密钥。

在程序示例中 w＝32，那么 u＝4，第 6 行代码到第 9 行代码实现了将每 4 个 unsigned char 型数据转化为 1 个 word 型数据，转化过程与 RC5 算法中的相应过程一致(见图 12-3)。第 10 行代码到第 18 行代码，则是根据 RC_6 算法的密钥生成方法生成密钥序列。

在生成密钥过程中使用了 rotL()函数用于循环移位，rotL()函数的详细代码见程序清单 13-3。

程序清单 13-3

```
word RC_6::rotL(word x,word y)
{
    return (((x)<< (y&(w- 1))) | ((x)>> (w- (y&(w- 1)))));
}
```

rotL()函数的循环左移的原理与程序清单 12-3 中的循环移位的原理和实现方法都相同。rotL()函数实现了循环左移的功能。

13.2.3 加密和解密的实现

RC6 的加密和解密过程通过函数 encryption()和 decryption()来实现，每次加密和解密过程都是针对 4 个 word 型数据进行，即每次处理 128 位数据。加密函数 encryption()的具体实现代码见程序清单 13-4。

程序清单 13-4

```
01 void RC_6::encryption(word in[],word out[])
02 {
03     word A,B,C,D;
04     A= in[0];
05     B= in[1]+ S[0];
06     C= in[2];
07     D= in[3]+ S[1];
08     int i;
09     for(i= 1;i< = 20;i+ + )
10     {
11         encHelper(i,A,B,C,D);
12     }
13     out[0]= A+ S[42];
14     out[1]= B;
15     out[2]= C+ S[43];
16     out[3]= D;
17 }
```

在程序清单 13-4 中，第 4 行代码到第 8 行代码是将输入的明文分别赋给 4 个 word 型变量，第 9 行代码到第 12 行代码是进行 20 轮加密，每轮加密过程通过函数 encHelper()来完成，第 13 行代码到第 16 行代码是将加密结果输出。

单轮加密函数 encHelper()的实现代码见程序清单 13-5。

程序清单 13-5

```
void RC_6::encHelper(int i,word &A,word &B,word &C,word &D)
{
    word u,t;
    u=rotL(D*(D+D+1),5);
    t=rotL(B*(B+B+1),5);
    A=rotL(A^t,u)+S[2*i];
    C=rotL(C^u,t)+S[2*i+1];
    word temp=A;
    A=B;
    B=C;
    C=D;
    D=temp;
}
```

encHelper()函数实现了两个基本功能，单轮加密和数据交换，将交换后的数据用于下一轮加密，在加密过程中还是用函数 rotL()来进行循环移位操作，rotL()函数为生成密钥过程中所使用的函数，见程序代码清单 13-3。

RC6 算法的解密过程是加密过程的逆过程，具体实现代码见程序清单 13-6。

程序清单 13-6

```
01 void RC_6::decryption(word in[],word out[])
02 {
03     word A,B,C,D;
04     A=in[0]-S[42];
05     B=in[1];
06     C=in[2]-S[43];
07     D=in[3];
08     int i;
09     for(i=20;i>=1;i--)
10     {
11         decHelper(i,D,A,B,C);
12     }
13     out[0]=A;
14     out[1]=B-S[0];
15     out[2]=C;
16     out[3]=D-S[1];
17 }
```

在解密过程中，首先将密文赋给 4 个 word 型变量，然后进行解密，第 9 行代码到第 12 行代码为 20 轮解密过程，在解密过程中使用了单轮解密函数 decHelper()函数来进行单轮解密，decHelper()函数的具体实现代码见程序清单 13-7。

程序清单 13-7

```
void RC_6::decHelper(int i,word &A,word &B,word &C,word &D)
```

```
{
    word u,t;
    u=rotL(D*(D+D+1),5);
    t=rotL(B*(B+B+1),5);
    C=rotR(C-S[2*i+1],t)^u;
    A=rotR(A-S[2*i],u)^t;
    word temp=D;
    D=C;
    C=B;
    B=A;
    A=temp;
}
```

decHelper()函数首先进行解密,然后进行数据交换用于下一步的解密,在解密过程中还使用了循环移位函数 rotR()来进行循环移位,函数 rotR()的具体代码可参考程序清单 12-6 完成,只需将函数名称改为 word RC_6::rotR(word x,word y)即可。

13.2.4 RC6 算法测试

RC6 算法的测试过程通过主函数 main()以及测试函数 test()来进行。通过主函数来设置用户密钥和明文,通过 test()函数来测试加密和解密过程。设置明文的函数为 setPlainText(),函数的具体代码见程序清单 13-8。

程序清单 13-8

```
void RC_6::setPlainText(word pText[])
{
    int i;
    for(i=0;i<4;i++)
    {
        plainText[i]=pText[i];
    }
}
```

setPlainText()函数的参数是输入的 word 型明文数组,函数实现的功能是将用户输入的明文数组赋给类明文数组,RC6 算法每次处理的数据为 4 个 word 型数据。

具体测试的过程通过函数 test()来完成,test()函数的具体代码见程序清单 13-9。

程序清单 13-9

```
void RC_6::test()
{
    int i;
    cout<<"The PlainText:"<<endl;
    for(i=0;i<4;i++)
    {
        cout<<hex<<plainText[i]<<" ";
    }
```

```
        cout<< endl;
        encryption(plainText,cipherText);
        cout<< "The CipherText:"<< endl;
        for(i= 0;i< 4;i+ + )
        {
            cout<< hex<< cipherText[i]<< " ";
        }
        cout<< endl;
        decryption(cipherText,deCipherText);
        cout<< "The deCipherText:"<< endl;
        for(i= 0;i< 4;i+ + )
        {
            cout<< hex<< deCipherText[i]<< " ";
        }
        cout<< endl;
    }
```

测试过程包括加密测试与解密测试两部分，具体测试过程的驱动通过主函数 main()函数来进行，main()函数的具体代码见程序清单 13-10。

程序清单 13-10

```
    int main()
    {
        RC_6 rc6;
        unsigned char key[b]= {  0x11,0x11,0x11,0x11,0x22,0x22,0x22,0x22,
                                 0x33,0x33,0x33,0x33,0x44,0x44,0x44,0x44};
        rc6.setKey(key);
        word pText[4];
        pText[0]= 0x12121212;
        pText[1]= 0x34343434;
        pText[2]= 0x45454545;
        pText[3]= 0x56565656;
        rc6.setPlainText(pText);
        rc6.test();
        return 0;
    }
```

main()函数主要实现输入用户密钥和明文，然后调用 RC_6 类的 test()函数来进行测试，具体测试结果如下：

```
The PlainText:
12121212 34343434 45454545 56565656
The CipherText:
87ca42e3 a0923af5 c016113d 94f9ca9b
The deCipherText:
12121212 34343434 45454545 56565656
```

在对明文加密后再进行解密得到解密后的明文与原先输入的明文一致。

13.3 习题与实践题

13.3.1 习题

1. 试详细说明 RC6 加密算法的加密和解密的基本过程。
2. 试详细说明 RC6 加密算法的密钥生成的基本原理。

13.3.2 实践题

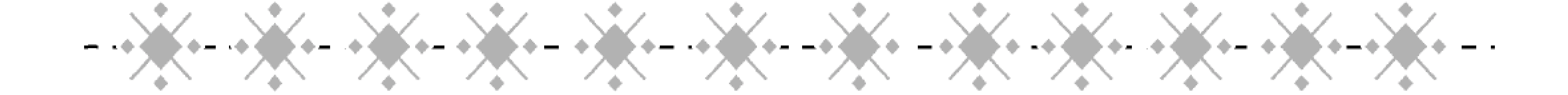
参考 13.2 节的关于 RC6 加密算法的实现过程,根据自己的理解编程完成 RC6 加密算法,要求:待加密的消息存储在相关文件中,加密后的数据保存到相关文件,并且具备检验解密是否正确的功能。

第 部分

公钥密码算法

公钥密码算法也称为非对称密码算法，公钥密码算法最早是由雷夫·莫寇(Ralph C. Merkle)在 1974 年提出来的，1976 年，狄菲(Whitfield Diffie)与赫尔曼(Martin Hellman)两位学者以单向陷门函数和单向暗门函数为基础，进行信息的发送与接收，至此，公钥密码算法开始走向实际应用。

公钥密码算法的基本特征是加密密钥可以公开传播而不会危及密码体制的安全性，从加密密钥计算得到解密密钥是难解的。例如，RSA 算法的安全性是基于大整数的素分解问题的难解性，ElGamal 公钥密码算法的安全性是基于有限域上离散对数问题的难解性，Menezes Vanstone 公钥算法的安全性是基于椭圆曲线上的离散对数问题的难解性。



第14章 RSA 算法

RSA 算法是公钥密码算法中最经典且应用最广泛的加密算法，在加密、数字签名和网络协议中具有广泛的应用，RSA 算法以大数运算为基础，其安全性主要依赖于对极大整数进行因数分解的难题。

14.1 基础知识

14.1.1 计算复杂性理论

计算复杂性理论是研究计算问题时所需要的时间和空间资源，对于密码学来说，计算复杂性理论提供了一种分析不同密码技术和算法的计算复杂性，通过对密码技术和算法的复杂性分析来确定相关密码技术与算法的安全性。

计算复杂性理论在密码学上主要研究算法在执行时所需要的计算资源和计算时间。

14.1.1.1 算法复杂性

算法复杂性是对算法计算所需要的时间和空间的一种度量，算法复杂性通常通过时间复杂性 T(time complexity)和空间复杂性 S(space complexity)两个变量来进行度量。

从算法的组成来看，算法主要由控制结构(如顺序结构、分支结构和循环结构等)和基本操作组成，算法的时间复杂性主要是指这两部分的综合效果，在进行算法比较时，通常会选择一种特定问题的基本操作来进行分析，用这个基本操作的执行次数作为算法的时间复杂性的度量。

算法时间复杂度通常是问题规模 n 的函数，假设算法中的基本操作的次数是问题规模 n 的某个函数 $f(n)$，那么，算法的时间度量可以描述为 $T(n)=O(f(n))$，即大 O 符号。$O(f(n))$ 称为算法的渐进时间复杂度，简称为时间复杂度。

示例 14-1 查找具有 n 个元素的一维数组中的最大元素的算法，可以采用依次遍历数组中的元素，并记下最大元素的下标，试计算该算法的时间复杂度。

解 问题的规模为 n，基本操作为比较，次数为 $f(n)=n$，所以时间复杂度 $T(n)=O(f(n))=O(n)$。

示例 14-2 假设某问题具有指数复杂性，$T(n)=6\times 2^n+n^2$，试计算该问题的时间复杂度。

解 当 $n\geqslant 4$ 时，$n^2\leqslant 2^n$，因此有 $T(n)\leqslant 6\times 2^n+2^n=7\times 2^n$，得 $T(n)=O(2^n)$。

常见的时间复杂度有：常数阶时间复杂度($O(1)$)、多项式阶时间复杂度(如 $O(n)$，$O(n^2)$)、指数阶时间复杂度(如 $O(2^n)$)和对数阶时间复杂度(如 $O(n\log n)$，$O(\log n)$)等。

算法的空间复杂度与时间复杂度类似，假设算法所需存储空间为问题规模 n 的某个函数 $f(n)$。那么算法的空间复杂度为 $S(n)=O(f(n))$。

目前，算法复杂度主要考虑算法的时间复杂度。

14.1.1.2 NP 问题简介

在解释 NP 问题之前，首先需要理解多项式时间、P 问题和非确定性问题。

多项式时间(polynomial time)　在计算复杂性理论中，是指一个问题的计算时间 $T(n)$ 不大于问题规模 n 的多项式倍数，$T(n)=O(n^k)$。

P 问题(polynomial)是指可以在多项式时间内被确定机(通常是指计算机)解决的问题，即存在多项式时间的算法的一类问题，称为 P 问题。

确定性问题通常是指只需要按照公式推导，按部就班一步一步执行就可以得到结果。例如，执行加、减、乘、除运算等。

非确定性问题是指无法按部就班地直接运算而得到结果。例如，将一个大合数分解成两个大素数的问题，现在还没有一个公式将一个大合数代入之后就可以得到两个大素数的公式。对于这类问题的答案是无法直接计算得到的，只能通过间接的“猜测”来得到结果，也就是非确定性问题。

非确定多项式(non-deterministic polynomial，NP)问题是指可以在多项式时间内被非确定机解决的问题。简单一点说：存在多项式时间的算法的一类问题，称之为 P 类问题；至今没有找到多项式时间算法解的一类问题，称之为 NP 问题。

14.1.2 中国剩余定理

中国剩余定理或孙子定理，最早见于《孙子算经》的“物不知数”问题：今有物不知其数，三三数之有二、五五数之有三，七七数之有二，问物有多少？这个问题的本质就是找出被 3，5 和 7 除时余数分别是 2，3 和 2 的所有整数 x。

定义 14.1　设 a,b,m 都是整数，如果 $m|(a-b)$，则称 a 和 b 模 m 同余，记为

$$a\equiv b(\mathrm{mod}\ m)$$

那么，将“物不知数”问题使用同余式组表示就是

$$\begin{cases}x\equiv 2(\mathrm{mod}\ 3)\\x\equiv 3(\mathrm{mod}\ 5)\\x\equiv 2(\mathrm{mod}\ 7)\end{cases}$$

定义 14.1 也可以描述为：如果 $a=b+kn$ 对某些整数 k 成立，那么 $a\equiv b(\mathrm{mod}\ n)$，如果 a 为正整数，b 为 0～n 之间的正整数，那么可以将 b 看做 a 被 n 整除后的余数，也称为 a 模 n 的余数，有时 a 也被称为 b 模 n 同余。

从 0 到 $n-1$ 的集合构成了模 n 的完全剩余集，也就是说，对每一个整数 a，它模 n 的余数是从 0 到 $n-1$ 的某个数。

模运算将计算结果限制在某一个范围之内，使用模运算会使得计算过程比较容易，模运算和普通运算一样具有可交换、可结合等特性，常用的模运算规律如下：

$$(a+b) \bmod n = ((a \bmod n)+(b \bmod n)) \bmod n$$
$$(a-b) \bmod n = ((a \bmod n)-(b \bmod n)) \bmod n$$
$$(a\times b) \bmod n = ((a \bmod n)\times(b \bmod n)) \bmod n$$
$$(a\times(b+c)) \bmod n = (((a\times b) \bmod n)+((a\times c) \bmod n)) \bmod n$$

定理 14.1　中国剩余定理设 $m_1,m_2,\cdots,m_r$ 是两两互素的正整数，$a_1,a_2,\cdots,a_r$ 是整数，则同余方程组

$$x\equiv a_i(\bmod m_i),\quad i=1,2,\cdots,r \tag{14-1}$$

模 $M=m_1m_2\cdots m_r$ 有唯一解

$$x=\sum_{i=1}^{r}a_iM_iy_i \bmod M \tag{14-2}$$

其中 $M_i=M/m_i, y_i=M_i^{-1} \bmod m_i, i=1,2,\cdots,r$。

对于“物不知数”问题有，$m_1=3,m_2=5,m_3=7,a_1=2,a_2=3,a_3=2$，故有

$$M=m_1m_2m_3=3\times5\times7=105$$
$$M_1=M/m_1=m_2\times m_3=5\times7=35$$
$$M_2=M/m_2=m_1\times m_3=3\times7=21$$
$$M_3=M/m_3=m_1\times m_2=3\times5=15$$
$$y_1=M_1^{-1}\bmod m_1=35^{-1}\bmod 3=2$$
$$y_2=M_2^{-1}\bmod m_2=21^{-1}\bmod 5=1$$
$$y_3=M_3^{-1}\bmod m_3=15^{-1}\bmod 7=1$$

从而得

$$\begin{aligned}x&=\sum_{i=1}^{3}a_iM_iy_i \bmod M\\&=(2\times35\times2+3\times21\times1+2\times15\times1)\bmod 105\\&=23\end{aligned}$$

因此，有“物不知数”问题的答案为 23。

14.1.3　Euler 函数

定义 14.2　Euler(欧拉)函数，也称为 Eulerφ 函数，记作 $\varphi(n)$，它表示小于 n 并与 n 互素的非负整数的个数。Euler 函数可以表示为

$$\varphi(n)=|\{x\mid 0\leqslant x\leqslant n-1,\gcd(x,n)=1\}|$$

如果 n 是一个素数，显然有 $\varphi(n)=n-1$。

定理 14.2　如果 $n=p\times q$，且 p 和 q 互素，那么 $\varphi(n)=(p-1)(q-1)$。

14.1.4　Euler 定理和 Fermat 小定理

定理 14.3(Euler 定理)　设 x 和 n 都是正整数，如果 $\gcd(x,n)=1$，则

$$x^{\varphi(n)}\equiv 1 \bmod n$$

根据 Euler 定理可以推广得到 Termat(费马)小定理。

推论 14.1　设 x 和 p 都是正整数，如果 p 是素数并且 $\gcd(x,p)=1$，则

$$x^{p-1}\equiv 1(\bmod p)$$

定理 14.4(Fermat 小定理) 设 x 和 p 都是正整数,如果 p 是素数,则

$$x^p \equiv x(\bmod\ p)$$

示例 14-3 计算 $3^{10\,000} \bmod 77$。

解 由 $\gcd(3,77)=1$,得 $\varphi(77)=\varphi(7\times 11)=6\times 10=60$,根据 Euler 定理得 $3^{60} \bmod 77=1$。因为

$$3^{10\,000} = 3^{166\times 60+40}$$

因此

$$3^{10\,000} \bmod 77 = 3^{166\times 60+40} \bmod 77 = 3^{40} \bmod 77 = 67$$

示例 14-4 计算 5 模 11 的乘法逆元。

解 由 11 为素数,得 $\varphi(11)=11-1=10$,由 Euler 定理推论得 5 模 11 的逆元是 5^9,因为

$$5^{10} \bmod 11 = 1$$

Euler 函数、Euler 定理和 Fermat 小定理在公钥密码算法中经常被用到。

14.1.5 模运算

在运算过程中合理使用模运算可以有效控制运算过程中的中间结果,对于任何一个长度为 k 的数模 n,其中间运算结果都不会超过 $2k$。模指数运算可以转换为一系列乘法运算,有效降低运算过程中的中间结果。

在进行模指数运算时,又可以将模指数运算分为两类,一类是 2 的幂次方的模运算,另一类不是 2 的幂次方的模运算。对于是 2 的幂次方的模运算比较简单,例如,要计算 $a^8 \bmod n$,并不需要进行 7 次乘法和一次模运算,而可以简化为 3 次较小的乘法和 3 次较小的模

$$(a\times a\times a\times a\times a\times a\times a\times a)\bmod n \to ((a^2 \bmod n)^2 \bmod n)^2 \bmod n$$

同样,对于 $a^{16} \bmod n$,计算过程可以转换为

$$a^{16} \bmod n \to (((a^2 \bmod n)^2 \bmod n)^2)^2 \bmod n$$

对于不是 2 的幂次方的模运算,其运算方法与是 2 的幂次方的模运算相似,只需要将指数转换为 2 的幂次方之和,转换的方法也比较简单,只需要将指数转换为二进制,再进行转换即可。例如,计算 $a^{27} \bmod n$,首先将 27 转换为二进制,27 的二进制表示为 11011,这样就将 $a^{27} \bmod n$ 的计算转换为

$$\begin{aligned} a^{27} \bmod n &= a^{(1+2+8+16)} \bmod n \\ &= (a\times a^2\times a^8\times a^{16}) \bmod n \\ &= ((a^2 \bmod n)\times(a^8 \bmod n)\times(a^{16} \bmod n)\times a) \bmod n \end{aligned}$$

这样就将不是 2 的幂次方的模运算转换为 2 的幂次方模运算。下面是进行该运算的 C++ 源程序。

程序清单 14-1

```
word modExp(word x,word y,word n)
{
    int s=1;
    while(y)
```

```
    {
        if(y&1)
        {
            s=(s*x)%n;
        }
        y>>=1;
        x=(x*x)%n;
    }
    return s;
}
```

程序清单 14-1 中使用的 word 型数据是为了方便处理，也可以直接使用 unsigned int。程序实现的功能是计算 $x^y \bmod n$，并返回计算结果。

以计算 $x^y \bmod n = 3^6 \bmod 11$ 为例，6 用二进制可以表示为$(110)_2$，那么 $6\&1=0$，$y=(y>>1)=3$，计算 $x=(x*x) \bmod n=(3*3) \bmod 11=9$；再计算 $y\&1=1$，则计算 $s=s*x \bmod n=1*9 \bmod 11=9$，$y>>1=1$，计算 $x=(x*x) \bmod n=(9*9) \bmod 11=4$；继续计算 $y\&1=1$，则计算 $s=s*x \bmod n=4*9 \bmod 11=3$；最后计算 $y>>1=0$，计算结束，得到 $3^6 \bmod 11=3$。该计算过程是效率较高的幂模运算方法，充分利用了幂模运算的特性。

14.2 素数与素性测试

素数又称质数，指在一个大于 1 的整数中，除了 1 和这个整数自身外，不能被其他整数整除的数。比 1 大，不是素数的数称为合数。素数的个数是无限的，在公钥密码学中常用到大的素数。

两个数互素是指：除了 1 之外没有公因子的两个整数称为互素，或者说，两个数的最大公因子为 1 的数称为互素。

检查一个正整数 N 是否是素数的最简单的方法是试除法，将该数 N 用小于等于 $\sqrt{N}$ 的所有素数去试除，若均无法整除，则 N 为素数。对于一个比较小的正整数，采用试除法是一个简单可行的方法，而对于大数，采用试除法不是可行的方法，例如：有一大数为 2^{1024} 大小，若采用试除法需检验 2^{512} 次，这在计算上是不可行的。

检验一个大数是否是素数，通常采用一些特定的检测方法，目前素数测试的方法包括两大类，一类是真素数测试法，一类是概率素数测试法。真素数测试法的速度通常比较慢，或者仅针对某一类素数进行测试，例如：Lucas-Lehmer 算法是针对 Mersenne 数的测试算法，这类素数测试方法在密码学的应用中受到了一定的限制。概率素数测试法比较实用，测试速度较快，常用的概率素数测试方法包括 Rabin-Miller 素性检测法、Solovag-Strassen 素性检测法和 Lehmann 素性检测法等。

基于概率的素数测试方法的基本思想为：给定一个正奇数 n，$Z(n)$为模 n 的非负整数集合，定义一个集合 $W(n)$属于 $Z(n)$并有如下属性：

(1) 给定一个属于集合 $Z(n)$的非负整数 a，可以在多项式时间内判断 a 是否属于集合 $W(n)$。

(2) 如果 n 是一个素数,那么 $Z(n)$ 中属于集合 $W(n)$ 的元素个数为 0。

(3) 如果 n 是一个合数,那么 $Z(n)$ 中属于集合 $W(n)$ 的元素个数大于等于 $n/2$。

如果 n 是一个合数,那么集合 $W(n)$ 中的元素称为合数 n 的证据,而集合 $L(n)=Z(n)-W(n)$ 中的元素称为合数 n 的伪证。

基于概率的素数测试算法的基本思路是:定义一个符合以上规则的集合 $W(n)$,对待测整数 n 随机地选择属于集合 $Z(n)$ 的元素 a,检测 a 是否属于集合 $W(n)$,如果 a 属于 $W(n)$,则可以确定 n 是一个合数,如果 a 不属于 $W(n)$,则 n 是素数的概率大于等于 1/2。对 n 随机地选择元素 a 进行 t 轮独立的测试,则 n 是素数的可能性可以被控制在 $1-(1/2)^t$ 以上。

14.2.1 Rabin-Miller 素性检测法

Rabin-Miller 是基于 Gary Miller 的基本思想,由 Michael Rabin 进行扩展得到的一个算法,Rabin-Miller 是 Fermat 小定理的一个变形改进,它的基础理论是由 Fermat 小定理引伸而来。

Rabin-Miller 素性检测法的基本方法为:要测试 n 是否为素数,首先要将 $n-1$ 分解为 2^kq,其中 k 是 2 整除 $n-1$ 的次数,即

$$n-1=2^kq$$

然后选取一个整数 a,$1<a<n-1$。再计算下述幂序列模 n 的余数:

$$a^q, a^{2q}, \cdots, a^{2^{k-1}q}, a^{2^kq} \tag{14-3}$$

根据 Fermat 小定理,若 n 是素数,则 $a^{2^kq} \bmod n=a^{n-1} \bmod n=1$。序列(14-3)中,可能在 a^{2^kq} 之前就存在余数为 1 的元素。将序列(14-3)表示为 $\{a^{2^jq}, 0\leqslant j\leqslant k\}$,那么,若 n 是素数,则有某个最小的 $j(0\leqslant j\leqslant k)$ 使得 $a^{2^jq} \bmod n=1$,分两种情况进行考虑:

当 $j=0$ 时有 $a^q-1 \bmod n=0$。

当 $1\leqslant j\leqslant k$ 时有 $(a^{2^jq}-1) \bmod n=(a^{2^{j-1}q}-1)(a^{2^{j-1}q}+1) \bmod n=0$,即 n 整除 $a^{2^{j-1}q}-1$ 或 $a^{2^{j-1}q}+1$,由于 j 是使 n 整除 $a^{2^jq}-1$ 的最小整数,因此 n 不能整除 $a^{2^{j-1}q}-1$,所以 n 整除 $a^{2^{j-1}q}+1$,即 $a^{2^{j-1}q} \bmod n=(-1) \bmod n=n-1$。

结论 如果 n 是素数,要么序列$(a^q, a^{2q}, \cdots, a^{2^{k-1}q}, a^{2^kq})$的第一个元素为 1,要么序列中的某元素为 $n-1$。

具体实现 Rabin-Miller 素性测试的过程如下:

(1) 选择一个待测随机数 n,计算 k(2 整除 $n-1$ 的次数)。

(2) 由 $n-1=2^kq$ 计算 q。

(3) 随机选取整数 a,$1<a<n-1$。

(4) 如果 $a^q \bmod n=1$,则 n 可能是素数。

(5) 对于 $1\leqslant j\leqslant k$,$a^{2^{j-1}q} \bmod n=n-1$,则 n 可能是素数。

(6) 否则 n 是合数。

示例 14-5 使用 Rabin-Miller 素性检测法,判断 37 是否是素数。

解 由 $n-1=37-1=36=2^2\times 9$,得 $k=2$,$q=9$。

根据 $1<a<n-1$:

选取 $a=20$,$20^9 \bmod 37=31$,既不为 1,也不为 36,继续测试。计算$(20^9)^2 \bmod 37=36$,因此测试返回可能是素数。

选取 $a=21$，$21^9 \bmod 37=36$，因此测试返回可能是素数。

对在 1～36 之间的所有整数执行测试算法，都返回可能是素数，这与 n 是素数是一致的。

示例 14-6 使用 Rabin-Miller 素性检测法，判断 45 是否是素数。

解 由 $n-1=45-1=44=2^2\times11$，得 $k=2$，$q=11$。

根据 $1<a<n-1$：

选取 $a=20$，$20^{11} \bmod 45=5$，既不为 1，也不为 44，继续测试。计算 $(20^{11})^2 \bmod 45=25$，同样既不为 1，也不为 44。

根据测试方法的第(5)条，已经对所有的 j 进行测试，最终返回是合数。

14.2.2 Solovag-Strassen 素性检测法

14.2.2.1 Solovag-Strassen 素性测试法基础

Solovag-Strassen 素性检测法是 Robert M. Solovag 和 Volker Strassen 开发的一种基于概率的基本检测法，在该算法中使用了 Jacobi(雅可比)函数来测试一个整数是否是素数。

定义 14.3 设 a 和 p 是两个整数，且 p 是素数。如果存在整数 r，使得 $a\equiv r^2 \pmod p$ 成立，则称 a 是模 p 的二次剩余。如果不存在整数 r，使得 $a\equiv r^2 \pmod p$ 成立，则称 a 是模 p 的二次非剩余。

例如：$p=5$，那么二次剩余是 1，4。

$$1^2=1\equiv1 \pmod 5,\quad 2^2=4\equiv4 \pmod 5,$$
$$3^2=9\equiv4 \pmod 5,\quad 4^2=16\equiv1 \pmod 5$$

没有 x 值可满足

$$x^2=2 \pmod 5,\quad x^2=3 \pmod 5$$

因此，对模 7 的二次非剩余是 2、3。

定义 14.4(Legendre(勒让德)符号) 对于整数 a 和素数 p，定义 legendre 符号 $\left(\frac{a}{p}\right)$ 如下：

$$\left(\frac{a}{p}\right)=\begin{cases}0, & \text{如果 } a\equiv 0 \bmod p\\ +1, & \text{如果不满足 } a\equiv 0 \bmod p\text{，且对于某个整数 } x, x^2\equiv a \pmod p\\ -1, & \text{如果不存在 } x\text{，使得 } x^2\equiv a \pmod p\end{cases}$$

当 $\left(\frac{a}{p}\right)=1$ 时，称 a 是模 p 的二次剩余，当 $\left(\frac{a}{p}\right)=-1$ 时，称 a 是模 p 的二次非剩余。

Legendre 符号记作 $L(a,p)$，Legendre 符号在实际计算过程中可以采用以下方法：

- 如果 a 被 p 整除，那么，$L(a,p)=0$。
- 如果 a 是对模 p 的二次剩余，那么，$L(a,p)=1$。
- 如果 a 是对模 p 的二次非剩余，那么，$L(a,p)=-1$。

定义 14.5(Jacobi 符号) Jacobi 符号是 Legendre 符号的推广，对于整数 a 和正奇数 n，$n\geqslant3$，n 的素分解为

$$n=p_1^{e_1}p_2^{e_2}\cdots p_r^{e_r}$$

其中 $p_i>2$，且为素数，$e_i\geqslant1$，$i=1,2,\cdots,r$，对于任意整数 a 有

$$\left(\frac{a}{n}\right)=\left(\frac{a}{p_1}\right)^{e_1}\left(\frac{a}{p_2}\right)^{e_2}\cdots\left(\frac{a}{p_r}\right)^{e_r}$$

称$\left(\frac{a}{n}\right)$为 Jacobi 符号。

Jacobi 符号具有如下性质：

(1) 如果 n 是素数，那么，Jacobi 符号$\left(\frac{a}{n}\right)$与对应的 Legendre 符号的含义相同。

(2) 如果 $a\equiv b(\bmod n)$，那么 $\left(\frac{a}{n}\right)=\left(\frac{b}{n}\right)$。

(3) $\left(\frac{a}{n}\right)=\begin{cases}0, & 若\ \gcd(a,n)\neq 1,\\ \pm 1, & 若\ \gcd(a,n)=1\end{cases}$。

(4) $\left(\frac{ab}{n}\right)=\left(\frac{a}{n}\right)\left(\frac{b}{n}\right)\rightarrow\left(\frac{a^2}{n}\right)=1$ 或 0。

(5) $\left(\frac{a}{mn}\right)=\left(\frac{a}{m}\right)\left(\frac{a}{n}\right)\rightarrow\left(\frac{a}{n^2}\right)=1$ 或 0。

(6) $\left(\frac{-1}{n}\right)=(-1)^{\frac{n-1}{2}}=\begin{cases}1, & 若\ n\equiv 1(\bmod 4),\\ -1, & 若\ n\equiv 3(\bmod 4)\end{cases}$。

(7) $\left(\frac{2}{n}\right)=(-1)^{\frac{n^2-1}{8}}=\begin{cases}1, & 若\ n\equiv 1,7(\bmod 8),\\ -1, & 若\ n\equiv 3,5(\bmod 8)\end{cases}$。

(8) 如果 a 和 b 是奇数，且 a 与 b 互素，那么

$$J(a,b)=(-1)^{\frac{(a-1)(b-1)}{4}}J(b,a)。$$

Jacobi 符号在计算中可以使用 $J(a,n)$来标记，在具体计算过程中可以使用以下规则：

- $J(0,n)=0$。
- $J(1,n)=1$。
- $J(a\times b,n)=J(a,n)\times J(b,n)$。
- 如果$(n^2-1)/8$ 是偶数，那么 $J(2,n)=1$，否则 $J(2,n)=-1$。
- $J(a,n)=J((a \bmod n),n)$。
- $J(a,b_1\times b_2)=J(a,b_1)\times J(a,b_2)$。
- 如果 a 和 b 都是奇数，且它们互素（最大公因子为 1），那么：如果$(a-1)(b-1)/4$ 是偶数，则 $J(a,b)=J(b,a)$，如果$(a-1)(b-1)/4$ 是奇数，则 $J(a,b)=-J(b,a)$。

定理 14.5（Euler 公式） 设 p 是一个素数，那么对于任意整数 a 有

$$\left(\frac{a}{p}\right)\equiv a^{\frac{p-1}{2}}\quad(\bmod p) \tag{14-4}$$

定理 14.6（Solovag Strassen 定理） 对于一个奇数 n，且 $n>2$，对于任意一个与 n 互素的整数 a，成立

$$\left(\frac{a}{n}\right)\equiv a^{\frac{n-1}{2}}(\bmod n) \tag{14-5}$$

14.2.2.2 Solovag-Strassen 素性测试法

设 p 是一个待测数，Solovag-Strassen 素性测试方法可以描述如下：

(1) 选择一个小于 p 的随机数 a。

(2) 计算 $\gcd(a,p)$，若 $\gcd(a,p)\neq 1$，则 a 与 p 不互素，所以 p 是合数，测试结束。

(3) 计算 $j=a^{(p-1)/2} \bmod p$。

(4) 计算 Jacobi 符号 $J(a,p)$。

(5) 如果 $j\neq J(a,p)$，那么 p 不是素数，测试结束。

(6) 如果 $j=J(a,p)$，那么 p 不是素数的可能性至多是 50%，重复到第一步测试，直到测试到所要求的概率结束。

Solovag-Strassen 素性测试次数 t 确定待测数 p 是合数的概率，在通过 t 次测试之后，p 是合数的概率小于 $1/2^t$。

示例 14-7 使用 Solovag-Strassen 素性检测法，判断 221 是否是素数。

解 $n=221$，随机选择一个数 $a=47<n$，计算：

$$j = a^{(n-1)/2} \bmod n = 47^{110} \bmod 221 = -1 \bmod 221$$

$$J(47,221) = \left(\frac{47}{221}\right) \bmod 221 = -1 \bmod 221$$

因此，47 是 221 为素数的伪证，221 不是素数的可能性至多是 50%，继续选择 $a=2<n$ 进行测试，计算：

$$j = a^{(n-1)/2} \bmod n = 2^{110} \bmod 221 = 30 \bmod 221$$

$$J(2,221) = \left(\frac{2}{221}\right) \bmod 221 = -1 \bmod 221$$

因此 2 是 221 不是素数的证据，所以 221 不是素数，$221=13\times 17$。

14.2.3 Lehmann 素性检测法

Lehmann 素性测试方法是 Lehmann 根据 Fermat 小定理推导出来的一种素性测试方法，假设 p 是一个素数，那么，有

$$a^{p-1} \equiv 1 \bmod p$$

将上式改写为

$$a^{p-1} - 1 \equiv 0 \bmod p$$

根据公式 $A^2-B^2=(A+B)(A-B)$，可以得到

$$a^{p-1} - 1 = (a^{(p-1)/2} - 1)(a^{(p-1)/2} + 1)$$

根据

$$p \mid (x \cdot y) \Rightarrow (p \mid x) \vee (p \mid y)$$

因此，如果 $a^{p-1}-1\equiv 0 \bmod p$ 成立，那么，下面方程同样成立：

$$a^{(p-1)/2} - 1 \equiv 0 \bmod p \Rightarrow a^{(p-1)/2} \equiv 1 \bmod p \Rightarrow a^{(p-1)/2} = 1 (\text{在 } Z_p \text{ 中})$$

$$a^{(p-1)/2} + 1 \equiv 0 \bmod p \Rightarrow a^{(p-1)/2} \equiv -1 \bmod p \Rightarrow a^{(p-1)/2} = -1 = p-1 (\text{在 } Z_p \text{ 中}) \qquad (14\text{-}6)$$

通过 k 次 Lehmann 素性测试的数是素数的概率为 $1-1/2^k$。

假设待测数为 p，那么 Lehmann 素性测试过程如下：

(1) 选择一个小于 p 的随机数 a。

(2) 计算 $a^{(p-1)/2} \bmod p$。

(3) 如果 $a^{(p-1)/2}\not\equiv 1 (\bmod p)$ 或 $a^{(p-1)/2}\not\equiv -1 (\bmod p)$，那么 p 是合数，测试结束。

(4) 如果 $a^{(p-1)/2}\equiv 1 (\bmod p)$ 或 $a^{(p-1)/2}\equiv -1 (\bmod p)$，那么 p 不是素数的可能性最多

是 50%。

重复上述测试过程一直到所需的测试概率结束。

示例 14-8 使用 Lehmann 素性检测法，判断 221 是否是素数。

解 $p=221$，随机选择一个 $a=3<p$，计算：

$$a^{(p-1)/2} \bmod p = 3^{110} \bmod 221 = 87 \not\equiv 1(\text{或} -1)$$

因此 p 不是素数。

示例 14-9 使用 Lehmann 素性检测法，判断 223 是否是素数。

解 $p=223$，随机选择一个 $a=3<p$，计算：

$$a^{(p-1)/2} \bmod p = 3^{111} \bmod 223 = 222 = 223 - 1,$$

因此 p 不是素数的可能性最多是 50%，再继续选择一个 $a=5<p$，计算：

$$a^{(p-1)/2} \bmod p = 5^{111} \bmod 223 = 222 = 223 - 1,$$

因此 p 是素数的可能性是 $1-1/2^2$。依次类推进行重复计算，以达到计算 p 是素数的概率要求为止。

14.2.4 AKS 素性检测法

AKS 素性检测法是由三位印度计算机科学家 Agrawal、Kayal 和 Saxena 共同研究得到的一种在多项式时间内确定一个给定的整数是素数还是合数的检测法，AKS 素性检测算法可以用于确定一般给定的整数是否是素数，不需要满足某些特定的条件，例如 Lucas-Lehme 素性检测法也可以高速判断给定的整数是否是素数，但 Lucas-Lehme 素性检测法仅对 Mersenne(梅森)素数有效，而类似的 Pepin 测试则仅对 Fermat 素数有效。

AKS 素性检测主要是基于以下定理。

定理 14.7 整数 $n(\geqslant 2)$是素数，当且仅当

$$(x+a)^n \equiv x^n + a \pmod n \qquad (14\text{-}7)$$

这个同余多项式对所有与 n 互素的整数 a 均成立。

但使用该同余多项式并不能使素性测试在多项式时间内完成，AKS 修改同余多项式，以降低计算过程的复杂度。

$$(x+a)^n \equiv x^n + a(\bmod x^{r-1}, n) \qquad (14\text{-}8)$$

(14-8)式与存在多项式 f 与 g，使得

$$(x+a)^n - (x^n + a) = nf + (x^r - 1)g$$

成立的意义是等同的。

假设待测数为 n，且 $n>1$，那么 AKS 素性测试过程如下：

(1) 对于整数 a,b，且有 $a>1,b>1$，如果 $n=ab$ 成立，则输出 n 是合数，测试结束。

(2) 寻找最小的 r，使得 $o_r(n)>\log^2 n$ 成立。

(3) 对于 $a\leqslant r$，使得 $1<\gcd(a,n)<n$，则输出 n 是合数，测试结束。

(4) 如果 $n\leqslant r$，则 n 是素数，测试结束。

(5) 从 $a=1$ 到$\lfloor\sqrt{\varphi(r)}\log n\rfloor$：

如果$(x+a)^n\neq x^n+a \ (\bmod (x^r-1,n))$，则输出 n 是合数，测试结束。

(6) 输出 n 是素数。

这里 $o_r(n)$表示 $n \bmod r$ 的阶。

14.3 大数运算

在各类计算机语言中，基本都规定了各自基础数据类型所能处理数据的长度，变量的使用范围也受到最大长度的限制，例如：在 C++ 中使用的整型数据的最大长度为 64 位，用 16 进制表达的最大数为 0xFFFFFFFFFFFFFFFF，用十进制表示则为 18446744073709551615。而这个等级大小的整数远远达不到公钥密码的一些算法的要求，例如，目前公钥算法的主流算法 RSA 至少需要 512 位，目前通常是建立在 1024 位基础上，并且还有继续增长的趋势，因此需要建立一些专门用于处理大数运算的库或类来解决这一问题。

14.3.1 大数运算的基本方法

大数运算通常是指超出编译器基础数据类型所能处理长度的整数运算，大数运算的处理方法主要有以下几种：

- 用字符串表示大数——将大数用十进制字符数组表示，然后模拟人工“竖式计算”的方法进行计算，这种运算方式的优点在于简单且便于理解，但是，它的缺点也显而易见，例如用这种方式表示 1024 位的大数，所需的十进制的位数达数百位，进行任何一种运算都需要在数百个数组元素上进行多重循环，同时还需要大量的临时存储空间来存储计算过程中的中间结果等，因此运行效率较低。
- 用二进制表示大数——通过使用计算机语言中的位运算和逻辑运算来进行处理，使用这种方法来处理大数运算，代码设计比较复杂，程序的可读性较差并难以理解。
- 用 n 进制表示大数——在使用数组来处理大数时，进制越大，数组的大小越小，这样可以有效降低运算过程的时间复杂度和空间复杂度，并提高运算效率。

使用 n 进制来进行大数运算是目前比较有效的一种方法，对于不同的语言环境和编译器，可以采用不同的进制来处理，例如，在 C++ 语言中，unsigned int 型数据的长度可以作为数组中每个元素的长度，在 32 位机器中，unsigned int 型数据的长度为 32 位，这样就可以使用 32 位长度的数据作为数组元素的程度，而 unsigned long long int 的长度为 64 位，正好用于计算过程中的中间数据的处理。

14.3.2 基于 32 位进制的大数运算方法

32 位进制计算的基本原理和日常使用的十进制的计算原理类似，十进制的计算方法为逢十进一，每个元素的取值范围是 0～9，而 32 位进制是逢 2^{32} 进一。每个单独元素的最大值为 0xFFFFFFFF，每个元素的取值范围为 0x00000000～0xFFFFFFFF。

使用 32 位进制来处理公钥密码算法可以有效地降低存储空间，例如，若处理的 RSA 算法的位数为 512 位，若用数组进行处理，并采用 32 位进制数据进行处理，那么，所使用数组的大小为 16，而采用二进制来处理则需要数组的大小 512，而采用十进制来处理数组的大小也大于 100。

使用数组和 32 位进制来处理大数的过程并不复杂，其基本的处理方法如图 14-1 所示。

在使用数组表示大数时，数组索引的最小值所在的元素是大数的最高位，数组索引的最大值所在的元素是大数的最低位。

A[0]	A[1]	A[2]	A[3]
0x456789AB	0x3456789A	0x23456789	0x12345678

B[0]	B[1]	B[2]	B[3]
0x12345678	0xFF456789	0x3456789A	0x456789AB

+ - * / %

C[0]	C[1]	C[2]	C[3]
?	?	?	?

图 14-1　大数运算基本原理示意图

除图 14-1 的表示方法外，还可以按照数组索引的高位表示大数的高位，数组索引的低位表示大数的低位，处理过程正好与图 14-1 的过程相反，但在计算机处理数据的时候比较方便，也更接近于计算机处理数据的方法。

14.3.2.1　加法运算

在进行加法运算时，如果两个操作数的值都大于 0，那么将两个大数从低位对齐，即从数组索引的最大值所在位置对齐，然后从大数的低位开始加起，例如，在图 14-1 中则应从 A[3]+B[3]开始，若 A[3]+B[3]>0xFFFFFFFF 则需要进位，假设进位为 carry，则 carry=1，否则 carry=0，在进行其他位置运算时，可以直接将 carry 添加到具体运算过程中，例如在计算 A[2]+B[2]时，就可以直接进行 A[2]+B[2]+carry 运算，再判断 carry 的值，并用于下一步的计算。在实际计算中也可以将 carry 设为 0，在具体运算中，每一步的运算都加上 carry。图 14-1 中加法的具体计算过程如下：

(1) A[3] + B[3] + carry = 0x12345678 + 0x456789AB + 0x0 = 0x579BE023 <0xFFFFFFFF
→C[3]=0x579BE023，carry=0

(2) A[2] + B[2] + carry = 0x23456789 + 0x3456789A + 0x0 = 0x579BE023 <0xFFFFFFFF
→C[2]=0x579BE023，carry=0

(3) A[1] + B[1] + carry = 0x3456789A + 0xFF456789 + 0x0 = 0x1339BE023 >0xFFFFFFFF
→C[1]=0x339BE023，carry=1

(4) A[0] + B[0] + carry = 0x456789AB + 0x12345678 + 0x1 = 0x579BE024 <0xFFFFFFFF
→C[0]=0x579BE024，carry=0

假设进行加法运算的数组长度相同，那么，上述加法运算过程用伪码可以描述为

```
carry:=0
for i from n-1 to 0
    result:=A[i]+B[i]+carry
    C[i]:=result%0x100000000
    carry:=result/0x100000000
```

```
endfor
```

在计算过程中用 result 来存储中间计算结果，result 的数据长度要大于 A[i]和 B[i]的长度，例如，A[i]和 B[i]的长度为 32 位，则 A[i]+B[i]的长度可能超过 32 位，此时，用 64 位的 result 来存储中间结果，则可以保证溢出位数据不会丢失。

14.3.2.2 减法运算

大数减法运算时减数需大于被减数，若被减数大于减数，则需要将减数与被减数进行交换，最后再处理符号即可，这样，计算的结果总大于 0。

在大数加法中使用进位(carry)来处理加法结果大于 0xFFFFFFFF 的情况，而在大数减法中则采用借位(borrow)来处理被减数数组元素大于减数数组元素的情况。borrow 可以初始化为 0，图 14-1 中减法的具体计算过程如下：

(1) A[3]<B[3]→C[3]=0x100000000+A[3]−B[3]−borrow
=0x100000000+0x12345678−0x456789AB−0x0=0xCCCCCCCD
borrow=1

(2) A[2]<B[2]→C[2]=0x100000000+A[2]−B[2]−borrow
=0x100000000+0x23456789−0x3456789A−0x1=0xEEEEEEEE
borrow=1

(3) A[1]<B[1]→C[1]=0x100000000+A[1]−B[1]−borrow
=0x100000000+0x3456789A−0xFF456789−0x1=0x35111110
borrow=1

(4) A[0]>B[0]→C[0]=A[0]−B[0]−borrow
=0x456789AB−0x12345678−0x1=0x33333332
borrow=0

假设 A>B，那么，上述减法运算过程用伪码可以描述为

```
borrow:=0
for i from n-1 to 0
    if A[i]-B[i]-borrow>=0
        C[i]:=A[i]-B[i]-borrow
        borrow:=0
    else
        C[i]:=0x100000000+A[i]-B[i]-borrow
    endif
endfor
```

14.3.2.3 乘法运算

乘法运算是以相乘的两个数是正整数为基础，大数的乘法运算可以参考“竖式运算”的方法来完成，上述两个数组用“竖式运算”方法的运算过程如下：

					A[0]	A[1]	A[2]	A[3]
*					B[0]	B[1]	B[2]	B[3]
=					A[0]B[3]	A[1]B[3]	A[2]B[3]	A[3]B[3]
+				A[0]B[2]	A[1]B[2]	A[2]B[2]	A[3]B[2]	
+			A[0]B[1]	A[1]B[1]	A[2]B[1]	A[3]B[1]		
+		A[0]B[0]	A[1]B[0]	A[2]B[0]	A[3]B[0]			
	C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]

运算结果存储在数组 C[]中,数组 C 索引的位置为数组 A 和数组 B 的索引之和加 1,即 i+j+1,进位部分则加到 C[i+j],C 数组的长度为 A 数组长度和 B 数组长度之和。C 数组的最高位用于处理 A[0]B[0]的进位。

假设数组 A 的长度为 p,数组 B 的长度为 q,那么,以数组形式处理的大数乘法的伪码如下:

```
for i from p-1 to 0
    for j from q-1 to 0
        result:=A[i]*B[j]
        C[i+j+1]+:=result%0x100000000
        C[i+j]+:=result/0x100000000
    endfor
endfor
```

伪码中的 result 用于处理中间结果,若数组 A 和数组 B 中的元素为 32 位长度,那么,result 的数据长度为 64 位,其目的是防止中间结果溢出,32 位数据与 32 位数据相乘,得到的中间结果的最大位数为 64 位。在计算过程中的进位项直接加到 C[i+j]中,而不再计算进位,这样,可以简化进位的计算。

14.3.2.4 除法运算

除法是四则运算中最复杂的运算,公钥加密算法中用到的除法其结果只能用整数表示,其本质是带余除法,它是基于以下基本关系。

设 $A,B\in$ ≫为整数,$B>0$,于是有唯一确定的整数 X 和 R,使得

$$A = X\cdot B + R,\quad 且\quad 0\leqslant R < B$$

其中 X 是商,R 是 A 除 B 的余数。

对于整数 A 和 B,带余除法最简单的实现方法是从被除数 A 中不断地减去除数 B,直到余数 R 小于除数,通过上述计算得到的结果如下:

$$X = \lfloor A/B \rfloor$$

$$R = A - \lfloor A/B \rfloor B$$

在运算过程中所用到的除法都是整数除法,若直接使用减法运算来实现除法运算的实现方法是一种效率很低的方法,在实际计算过程中可以首先确定 X 的范围,然后通过折半的方法快速获得最终的 X。这种计算方法与“试除”法的计算原理一致。

假设整数 A 的长度为 p,整数 B 的长度为 q,并有 $p>q$,那么,X 的长度范围在 $p-q$ 到 $p-q+1$ 之间,其具体值的范围是长度为 $p-q$ 的最小值和长度为 $p-q+1$ 的最大值之间。

使用十进制的计算过程很容易就理解 X 的取值范围,假设用一个 5 位数除一个 3 位数,可能得到的最大值是 99999/100,结果正好是 5−3+1 位数的最大值:999,可能得到的最小值是 10000/999,结果正好是 5−3 位数的最小值:10。

因此,在“试除”过程中所需要尝试的范围是非常有限,在确定范围后再进行“试除”可以有效提高计算速度,同时将除法运算转换为乘法运算。

设 X 为除法 A/B 计算结果的整数部分,R 为 A/B 的余数,max 为 A/B 可能的最大结果,min 为 A/B 可能的最小结果,那么除法运算的伪码如下:

```
while(true)
    X:= (max+min)/2
    if((A-B×X)<0)
        max:=X
        continue
    elseif((A-B×X)≥B)
        min:=X
        continue
    R:=A-B×X
endwhile
```

在伪码的计算过程中使用 X:=(A+B)/2,其中涉及的除法只需要将(A+B)右移一位就可以完成,因此整个计算过程中并没有用到实际意义上的除法,而其他相关运算则已经解决。

示例 14-10 试计算 28972÷347。

解 由算式可知 A=28972,B=347,p=5,q=3,则 max=999,min=10,计算如下:

$$=> X = (\text{min} + \text{max})/2 = 504 => R = A - X \times B =- 145916$$
$$=> R < 0 => \text{max} = X = 504$$
$$=> X = (\text{min} + \text{max})/2 = 257 => R = A - X \times B =- 60207$$
$$=> R < 0 => \text{max} = X = 257$$
$$=> X = (\text{min} + \text{max})/2 = 133 => R = A - X \times B =- 17179$$
$$=> R < 0 => \text{max} = X = 133$$
$$=> X = (\text{min} + \text{max})/2 = 71 => R = A - X \times B = 4335$$
$$=> R > B => \text{min} = X = 71$$
$$=> X = (\text{min} + \text{max})/2 = 102 => R = A - X \times B =- 6422$$
$$=> R < 0 => \text{max} = X = 102$$
$$=> X = (\text{min} + \text{max})/2 = 86 => R = A - X \times B =- 870$$
$$=> R < 0 => \text{max} = X = 86$$
$$=> X = (\text{min} + \text{max})/2 = 78 => R = A - X \times B = 1906$$
$$=> R > B => \text{min} = X = 78$$
$$=> X = (\text{min} + \text{max})/2 = 82 => R = A - X \times B = 518$$
$$=> R > B => \text{min} = X = 82$$
$$=> X = (\text{min} + \text{max})/2 = 84 => R = A - X \times B =- 176$$

$=> R < 0 => max = X = 84$

$=> X = (min + max)/2 = 83 => R = A - X \times B = 171$

$=> 0 < R < B => X = 83 \ R = 171$

得商为 83,余数为 171。

在除法运算过程中不仅得到了商,还得到了相应的余数。

14.4 RSA 公钥密码算法原理

RSA 公钥密码算法是一种经典的公钥算法,RSA 算法被广泛地应用于各种电子商务中。RSA 是 1977 年由在麻省理工学院工作的罗纳德·李维斯特(Ron Rivest)、阿迪·萨莫尔(Adi Shamir)和伦纳德·阿德曼(Leonard Adleman)一起提出的,以这三位科学家名字的开头来命名该算法。

RSA 算法的可靠性是基于大整数因式分解的难度,即大整数的分解越困难,那么,RSA 算法就越安全。一旦有人找到大整数因式分解的算法的话,RSA 算法的安全性基础就立刻瓦解。到目前为止,只要 RSA 算法使用的密钥有足够的长度,用 RSA 加密的信息是不能够被破解的。

在 RSA 加密算法中,明文以分组为单位进行加密。假设 p,q 为素数,并有 $n=pq$,那么,对明文分组 M 和密文分组 C,其加密和解密的过程如下:

$$C = M^e \bmod n$$
$$M = C^d \bmod n \qquad (14\text{-}9)$$

其中:(e,n)为加密密钥即公钥,(d,n)为解密密钥即私钥。

公钥和私钥的生成过程可以用以下方式描述:

(1) 选择两个素数 p 和 q。

- p 和 q 最好以随机方式进行选择,同时,p 和 q 的长度最好相同。素性测试可以采用已知的素性测试方法进行测试。

(2) 计算 $n=pq$。

- n 用于公钥和私钥的模运算,其单位通常用 bit 表示,n 的长度就是密钥长度。

(3) 根据欧拉函数计算:$\varphi(n)=\varphi(p)\varphi(q)=(p-1)(q-1)$。

(4) 选择一个整数 e,满足:$1<e<\varphi(n)$,且 $\gcd(e,\varphi(n))=1$,即 e 与 $\varphi(n)$互素。

- e 是 RSA 算法中的公钥。
- 最好选择具有较小汉明距离(非零元素的个数)的 e,这样可以有效提高加密的速度,例如选择 $2^{16}+1$,此时 e 的汉明距离为 1,同时,不要选择太小的数,选择太小的数会降低算法的安全性。

(5) 计算 d,$d^{-1} \equiv e(\bmod \varphi(n))$,$d$ 可以通过计算 $e(\bmod \varphi(n))$的乘法逆元来获得。

- 需要解决的计算为:$de \equiv 1 \pmod{\varphi(n)}$。
- 常用的计算方法为扩展的欧几里得算法。
- 计算得到的 d 为 RSA 算法的私钥。

通过计算得到的(e,n)为公钥,可以公开发布,计算得到的(d,n)为私钥,则需要秘密保

存。同时 p,q 和 $\varphi(n)$ 也需要秘密保存，通过 p,q 或 $\varphi(n)$ 可以直接计算得到私钥。

RSA 算法的加密和解密过程可以通过 Euler 定理的推论来证明。

上述 e 和 d 的选择满足

$$d \equiv e^{-1} \bmod (\varphi(n))$$

$$ed \equiv 1 \bmod (\varphi(n))$$

因此，ed 可以用 $k\varphi(n)+1$ 表示，这样我们可以得到

$$M^{k\varphi(n)+1} = M^{k(p-1)(q-1)+1} \equiv M \bmod n$$

即得

$$M^{ed} \equiv M \bmod n$$

这样就有

$$M = C^d \bmod n \equiv (M^e)^d \bmod n \equiv M^{ed} \bmod n \equiv M \bmod n$$

示例 14-11 设 $p=47, q=23, M=65$，试给出加密和解密的计算过程。

解 (1) $n=pq=47\times 23=1081$。

(2) $\varphi(n)=\varphi(pq)=\varphi(p)\varphi(q)=(p-1)(q-1)=46\times 22=1012$。

(3) 选择 e，有 $1<e<1012$，并与 1012 互素，选择 $e=2^8+1=257$，257 与 1012 互素。

(4) 计算 $e(\bmod \varphi(n))$ 的乘法逆元，得到 $d=949$。

(5) 加密 $C=M^e \bmod n=65^{257} \bmod 1081=871$。

(6) 解密 $M=C^d \bmod n=871^{949} \bmod 1081=65$。

在 RSA 算法的加密过程中要求被加密的明文的长度要小于 n，若被加密的明文的长度大于 n，则可以将明文分割成若干长度小于 n 的分组。同时，在加密过程中需要将要加密的字符转换成相应的编码，如 ASCII 码或 UNICODE 编码，并将转换后的数字组成一个数字，然后再进行加密。

14.5 RSA 公钥加密算法实现

RSA 加密算法的核心内容包括：大数运算、素性检测和 RSA 的加密与解密三个主要部分组成，同时通过利用大数的基本运算完成乘法逆元、幂模等运算。RSA 算法的实现以 32 位进制为基础来实现。

14.5.1 大数运算的实现

大数运算的实现是 RSA 算法实现的基础，大数运算的实现包括比较运算符的实现、基本运算的实现、模运算的实现、幂模运算的实现和乘法逆元计算的实现等几部分组成，在程序中通过 HugeInt 类来实现。HugeInt 类的基本组成见图 14-2。

HugeInt 类的声明见程序清单 14-2。

程序清单 14-2

```
typedef unsigned int word32;
typedef unsigned long long int word64;
const word64 w32= 0x100000000;
```

HugeInt	
- flag	: int
- v	: vector<word32>
+ HugeInt ()	
+ genHugeInt (int length)	: void
+ genSmallInt ()	: void
+ getLength (HugeInt hugeInt)	: int
+ &operator<< (ostream& output, const HugeInt & hugeInt)	: friend ostream
+ operator+ (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend HugeInt
+ operator- (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend HugeInt
+ operator* (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend HugeInt
+ operator/ (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend HugeInt
+ operator% (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend HugeInt
+ avg (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend HugeInt
+ shiftRightOne (const HugeInt & hugeInt)	: friend HugeInt
+ expMod (const HugeInt & x, const HugeInt & y, const HugeInt & n)	: friend HugeInt
+ invMod (HugeInt A, HugeInt B)	: friend ExtEuc
+ gcd (const HugeInt N, const HugeInt K)	: friend HugeInt
+ setFlag (int b)	: void
+ operator> (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend bool
+ operator>= (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend bool
+ operator< (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend bool
+ operator<= (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend bool
+ operator== (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend bool
+ operator!= (const HugeInt & hugeInt1, const HugeInt & hugeInt2)	: friend bool

图 14-2 HugeInt 类的基本组成

```
struct ExtEuc;
class HugeInt
{
    public:
        HugeInt();
        void genHugeInt(int length);
        void genSmallInt();
        void clearHugeInt();
        int getLength(HugeInt);
        friend ostream &operator<< (ostream&,const HugeInt&);
        friend HugeInt operator+ (const HugeInt&,const HugeInt&);
        friend HugeInt operator- (const HugeInt&,const HugeInt&);
        friend HugeInt operator * (const HugeInt&,const HugeInt&);
        friend HugeInt operator/(const HugeInt&,const HugeInt&);
        friend HugeInt operator%(const HugeInt&,const HugeInt&);
        friend HugeInt shiftRightOne(const HugeInt&);
        friend HugeInt expMod(const HugeInt&,const HugeInt&,const HugeInt&);
        friend ExtEuc invMod(HugeInt,HugeInt);
        friend HugeInt gcd(const HugeInt,const HugeInt);
        void setFlag(int);
        friend bool operator> (const HugeInt&,const HugeInt&);
        friend bool operator>= (const HugeInt&,const HugeInt&);
```

```
            friend bool operator< (const HugeInt&,const HugeInt&);
            friend bool operator<= (const HugeInt&,const HugeInt&);
            friend bool operator== (const HugeInt&,const HugeInt&);
            friend bool operator== (const HugeInt&,const HugeInt&);
            friend bool operator!= (const HugeInt&,const HugeInt&);
        private:
            int flag;
            vector<word32> v;
};
    struct ExtEuc
    {
        HugeInt d;
        HugeInt s;
        HugeInt t;
    };
```

类中的32位数据使用word32来处理，64位数据使用word64来处理，而w32则是用来处理进位和借位运算。变量v(vector)用来存储大数，采用向量来存储，在动态处理大数运算时更为方便，flag则是用来处理大数的符号。结构体ExtEuc则是在进行模逆运算时使用的结构体。

类HugeInt中各成员函数的作用如下：

- HugeInt()——构造函数，用于初始化flag。
- genHugeInt()——用于生成大数，函数的参数为所需大数的长度。
- genSmallInt()——用于生成小数，但生成的数仍使用向量表示，生成的数用于检验大数生成中所使用的各算法，该函数属于辅助函数，在最终算法实现中可以省略。
- clearHugeInt()——用于清空大数中的所有元素。
- getLength()——用于获取大数向量的大小，函数返回大数向量的大小。
- setFlag()——用于处理flag，即大数的正负。
- avg()——用于计算两个大数的平均值。
- shiftRightOne()——用于大数右移一位。
- expMod()——用于幂模运算。
- invMod()——用于计算模n的乘法逆元。
- gcd()——欧几里得函数。

除上述函数之外，其余函数分为两类：一类是常用运算符的重载，另一类是常用逻辑运算符重载。

14.5.1.1 初始化

初始化共有三部分组成：构造函数、大数生成和以大数形式表示的小数生成。构造函数主要初始化大数的正负符号，其完整代码见程序清单14-3。

程序清单 14-3

```
HugeInt::HugeInt()
```

```
{
    flag=1;
}
```

构造函数的作用是将 flag 设置为 1，即表示相应大数为正数。在运算过程中的 flag 是通过函数 setflag 来设置正负数的标记，函数代码见程序清单 14-4。

程序清单 14-4

```
void HugeInt::setFlag(int b)
{
    flag=b;
}
```

大数生成通过函数 genHugeInt(int length)来实现，函数具体代码见程序清单 14-5。

程序清单 14-5

```
void HugeInt::genHugeInt(int length)
{
    v.clear();
    srand((unsigned)time(0));
    int i;
    for(i=0;i<length;i++)
    {
        v.push_back(rand()+1);
        Sleep(100);
        v[i]|=(rand()+1)<<16;
    }
}
```

大数存储于向量，向量中的每个元素为 32 位大小，具体生成多少位的大数通过函数的参数来确定。例如，要生成 512 位的大数，则参数的大小为 16，16×32=512，这样，生成的大数正好为 512 位。同样，要生成 1024 位的大数，则参数的大小为 32。

大数生成通过随机函数来生成，主要方便用于大数生成，在大数生成过程中每次生成 16 位伪随机数，然后左移 16 位，再与下一次生成的 16 位伪随机数进行或运算，这样，就可以获得 32 位的大数元素。重复执行所设定的次数就可以获得所需长度的大数。

在数据初始化中，还有一个函数 genSmallInt()，该函数的作用是生成大数形式的小数，函数具体代码见程序清单 14-6。

程序清单 14-6

```
void HugeInt::genSmallInt()
{
    v.clear();
    int n;
    cout<<"输入小数：";
    cin>>n;
```

```
        v.push_back(n);
    }
```

函数 genSmallInt()的功能比较简单，其作用是通过键盘获得输入，并将获得的输入加入大数的向量中，该函数是用于检验算法的正确性。

genSmallInt()函数中小数的生成也可以通过函数的参数进行传递，只需要将函数的原型改为：void genSmallInt(int n)，具体实现的方法与程序清单 14-6 类似。

14.5.1.2 比较运算符重载的实现

大数的比较运算符主要作用是辅助各类大数运算的实现，共包括 6 种常用的比较运算符，实现的方法是重载。

“>”运算符重载的实现

大于运算符“>”重载函数的参数有两个，参数类型为 HugeInt，“>”重载函数的具体代码见程序清单 14-7。

程序清单 14-7

```
bool operator> (const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    if(hugeInt1.v.size()> hugeInt2.v.size())
    {
        return true;
    }
    if(hugeInt2.v.size()> hugeInt1.v.size())
    {
        return false;
    }
    int i;
    for(i=hugeInt1.v.size()-1;i>=0;i--)
    {
        if(hugeInt1.v[i]> hugeInt2.v[i])
        {
            return true;
        }
        else if(hugeInt2.v[i]> hugeInt1.v[i])
        {
            return false;
        }
    }
    return false;
}
```

在大数运算类的实现中采用了向量索引的高位代表大数的高位，向量索引的低位代表大数的低位，使得运算过程和普通的运算过程更为接近。

大于运算符的重载首先比较两个大数向量的长短，根据向量的长短来确定大数的大小。

在大数向量长度相同的情况下，通过从高位到低位逐个元素比较的方法来确定大数的大小，若两个大数元素大小相等则继续比较，若元素大小不同则根据元素大小情况返回对应的结果，若各个元素的大小都相同则返回 false。

“＞＝”运算符重载的实现

大于等于运算符“＞＝”重载函数的参数与大于重载函数的参数相同，实现方法与大于运算符重载的实现类似，大于等于运算符重载的具体代码见程序清单 14-8。

程序清单 14-8

```
bool operator>=(const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    if(hugeInt1.v.size()>hugeInt2.v.size())
    {
        return true;
    }
    if(hugeInt2.v.size()>hugeInt1.v.size())
    {
        return false;
    }
    int i;
    for(i=hugeInt1.v.size()-1;i>=0;i--)
    {
        if(hugeInt1.v[i]>hugeInt2.v[i])
        {
            return true;
        }
        else if(hugeInt2.v[i]>hugeInt1.v[i])
        {
            return false;
        }
    }
    return true;
}
```

大于等于运算符重载的实现方法与大于运算符重载的实现方法的不同之处在于当两个大数的各个元素完全相同时返回 true，其他各部分则完全相同，元素比较的方法也是从高位开始比较到低位。

“＜”运算符重载的实现

小于运算符“＜”重载的实现方法正好与大于运算符的实现方法相反，小于运算符重载的函数参数也是 HugeInt 型的两个参数，具体实现代码见程序清单 14-9。

程序清单 14-9

```
bool operator<(const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    if(hugeInt1.v.size()<hugeInt2.v.size())
```

```
    {
        return true;
    }
    if(hugeInt2.v.size()<hugeInt1.v.size())
    {
        return false;
    }
    int i;
    for(i=hugeInt1.v.size()-1;i>=0;i--)
    {
        if(hugeInt1.v[i]<hugeInt2.v[i])
        {
            return true;
        }
        else if(hugeInt2.v[i]<hugeInt1.v[i])
        {
            return false;
        }
    }
    return false;
}
```

小于运算符重载同样也是从大数向量的长度开始，在大数向量长度相同的情况下再比较向量中元素的大小。在小于运算符重载的实现过程中。需要注意的是当两个大数的各个元素都相同时，返回值是 false，这与大于运算符重载时类似。

“<=”运算符重载的实现

小于等于运算符“<=”重载与小于运算符重载类似，函数参数也是两个 HugeInt 型参数，具体实现代码见程序清单 14-10。

程序清单 14-10

```
bool operator<=(const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    if(hugeInt1.v.size()<hugeInt2.v.size())
    {
        return true;
    }
    if(hugeInt2.v.size()<hugeInt1.v.size())
    {
        return false;
    }
    int i;
    for(i=hugeInt1.v.size()-1;i>=0;i--)
    {
        if(hugeInt1.v[i]<hugeInt2.v[i])
        {
```

```
                return true;
            }
            else if(hugeInt2.v[i]<hugeInt1.v[i])
            {
                return false;
            }
        }
        return true;
}
```

在小于等于运算符重载中,首先根据大数向量的长度来确定大小,然后根据大数向量中的各个元素的大小来确定大数的大小,当大数向量中的各个元素都相等时,返回 true。

“==”运算符重载的实现

“==”运算符重载则要求大数向量的长度相等,且大数向量中的各个元素的大小相等,“==”运算符重载的具体代码见程序清单 14-11。

程序清单 14-11

```
bool operator== (const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
        if(hugeInt1.v.size()!=hugeInt2.v.size())
        {
            return false;
        }
        int i;
        for(i=hugeInt1.v.size()-1;i>=0;i--)
        {
            if(hugeInt1.v[i]!=hugeInt2.v[i])
            {
                return false;
            }
        }
        return true;
}
```

“==”运算符重载的实现方法比较简单,若大数向量的长度不同则返回 false,若大数向量对应元素的值不同则返回 false,否则返回 true。

“!=”运算符重载的实现

“!=”运算符重载主要用于判断两个大数不相等,与“==”运算符重载的运算过程正好相反,若向量的长度不相等则返回 true,若大数向量对应元素的值不相等则返回 true,否则返回 false。

“!=”运算符重载函数的具体代码见程序清单 14-12。

程序清单 14-12

```
bool operator!= (const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
```

```
    if(hugeInt1.v.size()!=hugeInt2.v.size())
    {
        return true;
    }
    int i;
    for(i=hugeInt1.v.size()-1;i>=0;i--)
    {
        if(hugeInt1.v[i]!=hugeInt2.v[i])
        {
            return true;
        }
    }
    return false;
}
```

14.5.1.3 基本运算符重载的实现

基本运算符重载包括"+、-、*、/、%"共 5 种,这 5 种基本运算为后续的大数运算的实现提供基础。这 5 种基本运算的实现原理见图 14-3。

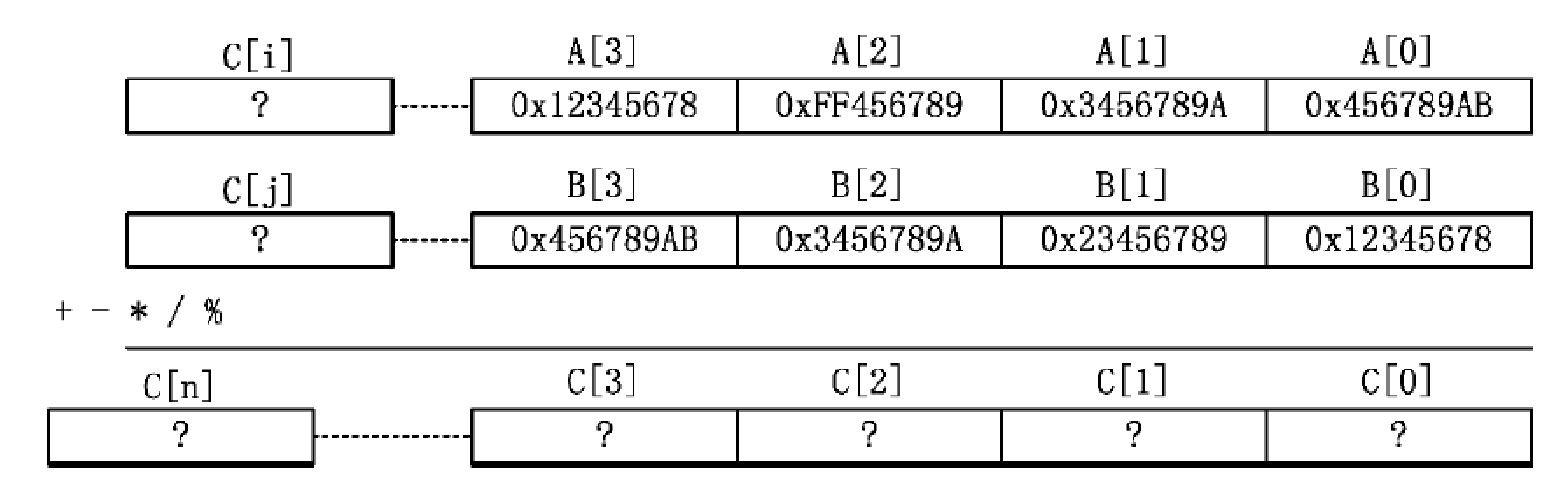


图 14-3 基本运算实现原理

在基本运算的实现中,大数向量索引的高位就代表大数的高位,大数向量索引的低位就代表大数的低位,这种实现方法能很方便地改变大数的位数,同时还很方便的清除高位的无效"0"数据。

加法运算的实现

"+"法运算符重载的参数是两个 HugeInt 型参数,函数的返回值类型也是 HugeInt 型,函数具体代码见程序清单 14-13。

程序清单 14-13

```
HugeInt operator+ (const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    HugeInt out;
    word64 temp;
    word32 carry=0;
    word32 i;
    if(hugeInt1>hugeInt2)
    {
```

```
        for(i=0;i<hugeInt2.v.size();i++)
        {
            temp=(word64)hugeInt1.v[i]+(word64)hugeInt2.v[i]+carry;
            if(temp>0xFFFFFFFF)
            {
                carry=1;
            }
            else
            {
                carry=0;
            }
            out.v.push_back((word32)temp);
        }
        for(i=hugeInt2.v.size();i<hugeInt1.v.size();i++)
        {
            temp=(word64)hugeInt1.v[i]+carry;
            if(temp>0xFFFFFFFF)
            {
                carry=1;
            }
            else
            {
                carry=0;
            }
            out.v.push_back((word32)temp);
        }
    }
    else
    {
        for(i=0;i<hugeInt1.v.size();i++)
        {
            temp=(word64)hugeInt1.v[i]+(word64)hugeInt2.v[i]+carry;
            if(temp>0xFFFFFFFF)
            {
                carry=1;
            }
            else
            {
                carry=0;
            }
            out.v.push_back((word32)temp);
        }
        for(i=hugeInt1.v.size();i<hugeInt2.v.size();i++)
        {
            temp=(word64)hugeInt2.v[i]+carry;
```

```
54                  if(temp>0xFFFFFFFF)
55                  {
56                      carry=1;
57                  }
58                  else
59                  {
60                      carry=0;
61                  }
62                  out.v.push_back((word32)temp);
63              }
64          }
65          if(carry==1)
66          {
67              out.v.push_back(carry);
68          }
69          return out;
70      }
```

“+”运算的实现在总体上分成三部分，第 7 行到第 35 行为第一部分，这部分是处理第 1 个参数大于第 2 个参数的情况，第 36 行到第 64 行为第二部分，这部分是处理第 2 个参数大于第 1 个参数的情况，第 65 行到第 68 行处理最后的进位。

在大数运算过程中，两个 32 位的数相加，其结果可能大于 32 位，因此中间结果需存放在大于 32 位的数据中，在程序中，是将中间结果存放在 64 位数据 temp 中，将 64 位的 temp 与 0xFFFFFFFF 进行比较，若大于 0xFFFFFFFF 则取进位 carry 为 1，低 32 位则为加法运算结果，加入大数向量。同时，在加法运算过程中，还需要将 32 位的向量数据强制转换为 64 位数据，这样，可以防止计算过程中的精度损失。

前两部分的计算都是以较小的数据的向量长度为界进行计算，向量长度较长的大数的余下部分则与进位 carry 进行加法运算，最后，若进位 carry 不为 0，则在大数向量的尾部加上最后一个向量元素。

减法运算的实现

“—”运算符重载的参数也是两个 HugeInt 型参数，函数的返回值类型也是 HugeInt 型，函数具体代码见程序清单 14-14。

程序清单 14-14

```
HugeInt operator-(const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    HugeInt out;
    word32 borrow=0;
    word64 temp;
    word32 i;
    HugeInt zero;
    zero.v.push_back(0);
    if(hugeInt1==hugeInt2)
    {
```

```
        return zero;
    }
    if(hugeInt1>hugeInt2)
    {
        out.setFlag(1);
        for(i=0;i<hugeInt2.v.size();i++)
        {
            if(hugeInt1.v[i]<(hugeInt2.v[i]+borrow))
            {
                temp=w32+(word64)hugeInt1.v[i]-(word64)hugeInt2.v[i]-borrow;
                borrow=1;
                out.v.push_back((word32)temp);
            }
            else
            {
                temp=(word64)hugeInt1.v[i]-hugeInt2.v[i]-borrow;
                borrow=0;
                out.v.push_back((word32)temp);
            }
        }
        for(i=hugeInt2.v.size();i<hugeInt1.v.size();i++)
        {
            if(hugeInt1.v[i]<borrow)
            {
                temp=w32+(word64)hugeInt1.v[i]-borrow;
                borrow=1;
                out.v.push_back((word32)temp);
            }
            else
            {
                temp=(word64)hugeInt1.v[i]-borrow;
                borrow=0;
                out.v.push_back((word32)temp);
            }
        }
    }
    else
    {
        out.setFlag(-1);
        for(i=0;i<hugeInt1.v.size();i++)
        {
            if(hugeInt2.v[i]<(hugeInt1.v[i]+borrow))
            {
                temp=w32+(word64)hugeInt2.v[i]-(word64)hugeInt1.v[i]-borrow;
                borrow=1;
```

```
                out.v.push_back((word32)temp);
            }
            else
            {
                temp=hugeInt2.v[i]-hugeInt1.v[i]-borrow;
                borrow=0;
                out.v.push_back((word32)temp);
            }
        }
        for(i=hugeInt1.v.size();i<hugeInt2.v.size();i++)
        {
            if(hugeInt2.v[i]<borrow)
            {
                temp=w32+hugeInt2.v[i]-borrow;
                borrow=1;
                out.v.push_back((word32)temp);
            }
            else
            {
                temp=hugeInt2.v[i]-borrow;
                borrow=0;
                out.v.push_back((word32)temp);
            }
        }
    }
    while(true)
    {
        if(out.v[out.v.size()-1]!=0)
        {
            break;
        }
        else
        {
            out.v.erase(out.v.end()-1);
        }
    }
    return out;
}
```

大数减法分为三种情况，第 1 种情况是两个大数相等，此时返回 0，但必须注意返回的"0"是以大数形式表示的"0"，即大数向量只有一个元素，这个元素的值为 0。第 2 种情况是函数的第 1 个参数的值大于第 2 个参数的值，此时只需按照正常减法进行运算，对大数向量从低位开始逐个运算。第 3 种情况是函数的第 2 个参数大于第 1 个参数，此时的运算过程是用第 2 个参数的值减去第 1 个参数的值，同时设置 flag 为－1，表示计算的结果为负值。

在大数减法运算过程中需要用到借位 borrow 来处理减数大于被减数的问题，此时就

需要使用超过 32 位的 w64 来处理减法，因此减法的中间结果 temp 也使用 64 位的数据来处理。

在加法运算过程中还可能产生高位数据为“0”的情况，此时就需要将大数向量的“0”元素清除，否则，会影响比较运算符的运算，在大数比较运算时通常是先比较大数向量的长度，然后再比较相应的值。因此，在减法运算结束时需检查高位是否为“0”，若高位为“0”则删除该向量元素，若高位不为 0，则结束清除高位为“0”的任务，具体代码见程序清单的第 81 行到 91 行。

乘法运算的实现

“*”运算符重载是实现两个大数的乘法，函数的两个参数也是 HugeInt 型，函数的返回类型也是 HugeInt，“*”运算符重载实现的具体代码见程序清单 14-15。

程序清单 14-15

```
HugeInt operator* (const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    HugeInt out;
    word64 temp;
    word32 carry=0;
    word32 i,j;
    for(i=0;i<hugeInt1.v.size();i++)
    {
        for(j=0;j<hugeInt2.v.size();j++)
        {
            temp=(word64)hugeInt1.v[i]*(word64)hugeInt2.v[j]+carry;
            carry=temp/w32;
            temp=temp&0xFFFFFFFF;
            if((i+j)>(out.v.size()-1)||out.v.size()==0)
            {
                out.v.push_back((word32)temp);
            }
            else
            {
                if(((word64)temp+(word64)out.v[i+j])>0xFFFFFFFF)
                {
                    carry+=1;
                    out.v[i+j]=(word32)(temp+out.v[i+j]);
                }
                else
                {
                    out.v[i+j]=(word32)(temp+out.v[i+j]);
                }
            }
        }
        if(carry>0)
        {
```

```
                out.v.push_back(carry);
                carry=0;
            }
        }
        return out;
}
```

在大数乘法运算过程中同样需要注意进位 carry 的问题，因此乘法中间运算的结果同样需要 64 位的变量来存储。

在大数乘法运算过程中还需要注意乘法运算的结果所存放的位置是否已经有数据，若有数据则和原来的数据进行加法运算，再存入对应的位置，并正确处理进位问题。

例如，在图 14-4 中，向量 A 和向量 B 进行乘法运算，当 B[0]与向量 A 的各个元素相乘时，只需将计算的结果直接加入向量的尾部即可，但在进行第 2 轮运算时，则需要考虑存储乘法结果的向量是否在相应的位置已经有元素，例如，第 2 轮运算的 A[0]B[1]的计算结果所存放的位置已经有前一轮的计算结果 A[1]B[0]，因此，A[0]B[1]的结果不是直接添加到向量，而是与 A[1]B[0]进行相加，然后将计算结果替换原来位置的值，此时同样需要注意进位的问题，若相加结果大于 0xFFFFFFFF，则 carry 加 1。

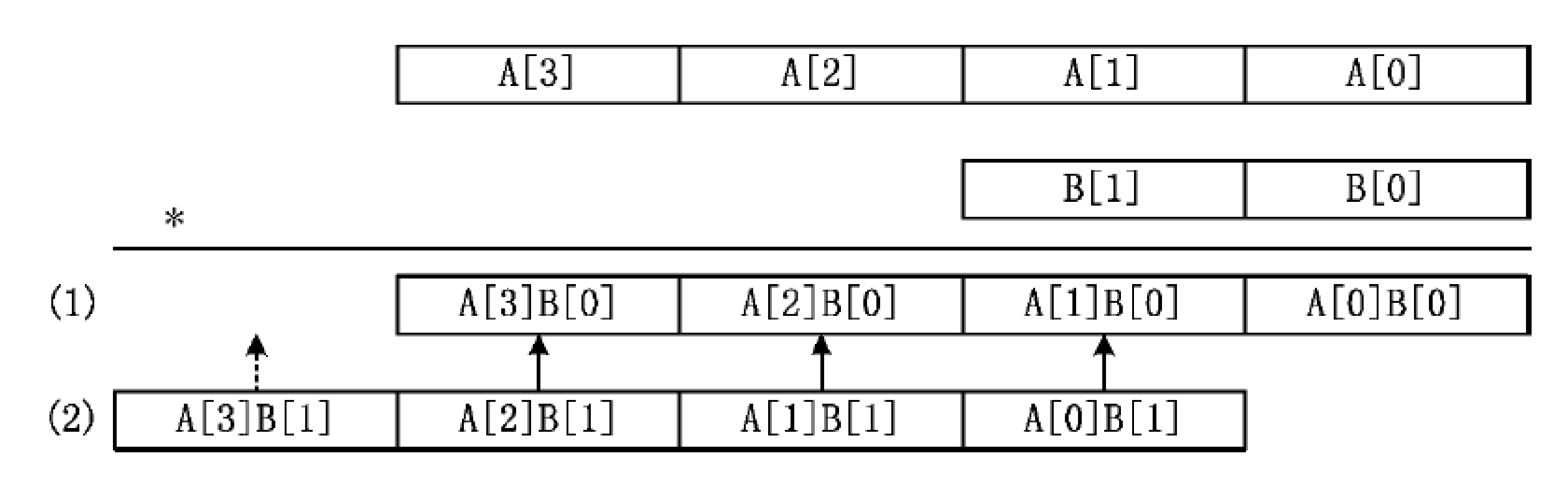


图 14-4　乘法运算过程示意图

在全部乘法运算完成之后，还需要注意最后一次进位的问题，当最后一次计算得到的 carry 大于“0”时，需要将 carry 添加到向量的最后。

除法运算的实现

“/”运算符重载实现两个大数相除，两个大数相除使用的方法是试乘法，通过获得两个大数的向量的大小来确定可能结果的最大值和最小值，在计算最大值和最小值的平均值来试乘，然后根据试乘的结果重新确定最大值和最小值，再计算平均值，依次重复至满足相应的条件。大数除法的详细代码见程序清单 14-16。

程序清单 14-16

```
HugeInt operator/(const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    HugeInt out;
    HugeInt max,min,X,R;
    word32 i;
    if(hugeInt1< hugeInt2)
    {
        out.v.push_back(0);
```

```
09          return out;
10      }
11      for(i=0;i<(hugeInt1.v.size()-hugeInt2.v.size()+1);i++)
12      {
13          max.v.push_back(0xFFFFFFFF);
14      }
15      for(i=0;i<(hugeInt1.v.size()-hugeInt2.v.size());i++)
16      {
17          min.v.push_back(0x0);
18      }
19      while(1)
20      {
21          X=avg(max,min);
22          R=hugeInt1-X*hugeInt2;
23          if(R.flag==-1)
24          {
25              max=X;
26          }
27          else if(R>=hugeInt2)
28          {
29              min=X;
30          }
31          else
32          {
33              break;
34          }
35      }
36      return X;
37  }
```

大数除法运算函数的参数为两个 HugeInt 型大数，函数的返回类型也是 HugeInt 型，大数除法的运算过程为先比较两个大数的大小，若参数 1 小于参数 2，则返回 0，否则结束。在后续计算过程中先确定除法运算结果的范围，见代码行第 11 行到第 18 行，即获得可能的最大值 max 和可能的最小值 min，接下来就进行循环计算。接下来就是循环计算，余数的计算方法为 R＝hugeInt1－X＊hugeInt2，当余数大于 0，且小于第 2 个大数时计算结束，其中 X 就是除法运算的结果。当余数小于 0 时，将 max 设为 x，当余数大于等于 hugeInt2 时，就将 min 设为 X，而每次运算 X 的取值为(min＋max)/2。(min＋max)/2 通过函数 avg()来完成，avg()函数的详细代码见程序清单 14-17。

程序清单 14-17

```
HugeInt avg(const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    HugeInt out;
    HugeInt temp;
    temp=hugeInt1+hugeInt2;
```

```
    word32 shiftCarry=0;
    word32 i;
    out.v.push_back(temp.v[0]/2);
    for(i=1;i<temp.v.size();i++)
    {
        shiftCarry=temp.v[i]&0x1;
        if(shiftCarry==1)
        {
            out.v[i-1]+=w32/2;
        }
        out.v.push_back(temp.v[i]/2);
    }
    if(out.v[out.v.size()-1]==0)
    {
        out.v.erase(out.v.end()-1);
    }
    return out;
}
```

计算平均值的本质就是将大数和除以 2,使用向量计算的基本原理见图 14-5。

	A[3]	A[2]	A[1]	A[0]
	11111001	11000011	10000111	10011101
A/2	01111100 1->	01100001 1->	01000011 1->	01001110 1×
	01111100	11100001	11000011	11001110

图 14-5 平均值计算原理

图 14-5 是以 8 位数据来描述平均值计算的原理,计算方法是从大数索引的低位计算到大数索引的高位,在计算 A[0]时只需直接除以 2,或右移一位。在计算 A[1]时,由于 A[1]的最低位为 1,因此需要转换为 A[0]的最高位,转换的方法可以将 A[0]的计算结果与 $10000000_{(2)}$进行或运算,或者直接加 $10000000_{(2)}$,两种方法具有相同的含义,其本质上就是低位向量元素的最高位就是高一位元素的最低位。

对于计算结果,同样需要注意最高位向量元素为"0"的情况,当最高位向量元素为"0"时,则需要将向量元素的最高位清除。

取余运算的实现

大数取余(%)运算实际上就是模运算,大数取余运算的实现方法和大数除法运算的实现方法相同,大数除法运算的返回结果为商,而大数取余运算的返回结果为余数。大数取余运算实现的具体代码见程序清单 14-18。

程序清单 14-18

```
HugeInt operator%(const HugeInt &hugeInt1,const HugeInt &hugeInt2)
{
    HugeInt out;
    HugeInt zero;
```

```
05      zero.v.push_back(0);
06      HugeInt max,min,X,R;
07      word32 i;
08      if(hugeInt1<hugeInt2)
09      {
10          out.v.push_back(0);
11          return hugeInt1;
12      }
13      if(hugeInt2==zero)
14      {
15          return zero;
16      }
17      for(i=0;i<(hugeInt1.v.size()-hugeInt2.v.size()+1);i++)
18      {
19          max.v.push_back(0xFFFFFFFF);
20      }
21      for(i=0;i<(hugeInt1.v.size()-hugeInt2.v.size());i++)
22      {
23          min.v.push_back(0x0);
24      }
25      while(1)
26      {
27          X=avg(max,min);
28          R=hugeInt1-X*hugeInt2;
29          if(R.flag==-1)
30          {
31              max=X;
32          }
33          else if(R>=hugeInt2)
34          {
35              min=X;
36          }
37          else
38          {
39              break;
40          }
41      }
42      return R;
43  }
```

大数取余运算的计算过程类似于大数的除法运算，在第 8 行到第 12 行代码行中，当大数 1 小于大数 2 时，返回的余数为大数 1，同时最后的返回值也为余数。

14.5.1.4 其他大数算法的实现

常用的大数运算除了比较运算、基本运算外，还有幂模运算、乘法逆元运算和欧几里得

(gcd)运算等,这些运算也是公钥密码算法中常用的运算方法,这些算法的实现方法都是基于大数基本运算来实现的。

幂模运算的实现

幂模运算主要用于计算 $A^B \bmod N$,幂模运算的基本原理在 14.1.5 节中有详细描述,幂模运算的具体实现见代码清单 14-19。

程序清单 14-19

```
HugeInt expMod(const HugeInt &x,const HugeInt &y,const HugeInt &n)
{
    HugeInt out;
    HugeInt zero;
    HugeInt X=x;
    HugeInt Y=y;
    out.v.push_back(1);
    zero.v.push_back(0);
    while(Y>zero)
    {
        if((Y.v[0])&1)
        {
            out=(out * X)%n;
        }
        Y=shiftRightOne(Y);
        X=(X * X)%n;
    }
    return out;
}
```

大数的幂模运算过程和普通整型数据的幂模运算过程完全一致,在运算过程中的右移一位的过程通过函数 shiftRightOne()来完成,shiftRightOne()函数的详细代码见程序清单 14-20。

程序清单 14-20

```
HugeInt shiftRightOne(const HugeInt& hugeInt)
{
    HugeInt out;
    HugeInt one,zero;
    zero.v.push_back(0);
    one.v.push_back(1);
    if(hugeInt==zero || hugeInt==one)
    {
        return zero;
    }
    word32 shiftCarry=0;
    word32 i;
    out.v.push_back(hugeInt.v[0]/2);
```

```
14        for(i=1;i<hugeInt.v.size();i++)
15        {
16            shiftCarry=hugeInt.v[i]&0x1;
17            if(shiftCarry==1)
18            {
19                out.v[i-1]+=w32/2;
20            }
21            out.v.push_back(hugeInt.v[i]/2);
22        }
23        if(out.v[out.v.size()-1]==0)
24        {
25            out.v.erase(out.v.end()-1);
26        }
27        return out;
28    }
```

右移一位函数的实现方法与在除法运算中使用的计算平均值函数 avg()的实现方法一致，avg()函数的参数包含两个 HugeInt 型参数，先计算和，再计算平均值，而计算两个大数平均值的本质就是将大数右移一位。在计算过程中当输入的大数为“0”或“1”时，直接返回“0”，在右移一位的运算过程中同样需要处理大数向量高位为“0”的问题，代码行第 16 行到第 20 行就是用于解决大数向量高位为“0”的问题。

欧几里得算法的实现

大数欧几里得算法的实现方法与 2.3.3 节的乘数密码算法实现中的欧几里得算法的实现方法一致，只是由原来的普通整数的计算转换为大数的计算。大数欧几里得算法实现的具体代码见程序清单 14-21。

程序清单 14-21

```
HugeInt gcd(const HugeInt N,const HugeInt K)
{
    HugeInt zero;
    HugeInt n=N;
    HugeInt k=K;
    zero.v.push_back(0);
    if(n==zero)
    {
        return k;
    }
    if(k==zero)
    {
        return n;
    }
    n=n%k;
    return gcd(k,n);
}
```

程序清单的 zero 是只有一个“0”元素的大数，其作用是用于判断 n 或 k 是否为 0。

乘法逆元计算的实现

在 RSA 加密算法和其他公钥密码算法中经常会在计算密钥时使用乘法逆元，乘法逆元的计算原理与实现方法与 2.3.2 节的扩展的欧几里得算法中所描述的方法一致，在实现过程中也是将普通整数改为大数，乘法逆元计算过程中所用到的结构体的声明如下：

```
struct ExtEuc
{
    HugeInt d;
    HugeInt s;
    HugeInt t;
};
```

该结构体中的数据为扩展的欧几里得算法的实现中所需要的相关数据，其中 s 为乘法逆元，具体原理见 2.3.2 节，乘法逆元计算的具体代码见程序清单 14-22。

程序清单 14-22

```
ExtEuc invMod(HugeInt A,HugeInt B)
{
    ExtEuc eu,eu1;
    HugeInt a=A;
    HugeInt b=B;
    HugeInt out;
    HugeInt zero,one;
    zero.v.push_back(0);
    one.v.push_back(1);
    if(b==zero)
    {
        eu1.d=a;
        eu1.s=one;
        eu1.t=zero;
        return eu1;
    }
    eu1=invMod(b,a%b);
    eu.d=eu1.d;
    eu.d.setFlag(eu1.d.flag);
    eu.s=eu1.t;
    eu.s.setFlag(eu1.t.flag);
    if(eu1.s.flag!=1 && eu1.t.flag!=1)
    {
        eu.t=(a/b)*eu1.t-eu1.s;
        eu.t.setFlag(-1);
    }
    else if(eu1.s.flag==1 && eu1.t.flag==1)
    {
```

```
29            eu.t=eu1.s-(a/b)*eu1.t;
30        }
31        else if(eu1.s.flag==1 && eu1.t.flag!=1)
32        {
33            eu.t=eu1.s+(a/b)*eu1.t;
34        }
35        else
36        {
37            eu.t=zero-((a/b)*eu1.t+eu1.s);
38        }
39        return eu;
40    }
```

在程序清单的第 22 行代码到第 38 行代码的计算过程是要计算 eu.t=eu1.s-(a/b)*eu1.t，但是由于大数的符号是通过专用的符号来表示的，因此在计算过程中需要考虑符号的问题，例如，当 eu1.s 为正数，eu1.t 为负数时，由于在减法运算中并不考虑正负数问题，因此，算式 eu.t=eu1.s-(a/b)*eu1.t 的计算需转换为 eu.t=eu1.s+(a/b)*eu1.t，其他各部分的计算转换原理与此相同。

14.5.2 素性检测的实现

素性检测过程的实现既可以作为 HugeInt 类的一部分，也可以单独作为一个类来实现，在本示例中素性检测由素性检测类 Prime 来实现，Prime 类的基本结构见图 14-6。

Prime	
+ calcK (const HugeInt& hugeInt)	: int
+ calcQ (const HugeInt& hugeInt)	: HugeInt
+ witness (const HugeInt& a, const HugeInt& n)	: bool
+ millerRabin (const HugeInt& n, int s)	: bool

图 14-6 Prime 类图

Prime 类的声明见程序清单 14-23。

程序清单 14-23

```
class Prime:public HugeInt
{
    public:
        int calcK(const HugeInt& hugeInt);
        HugeInt calcQ(const HugeInt& hugeInt);
        bool witness(const HugeInt& a,const HugeInt &n);
        bool millerRabin(HugeInt& n,int s);
};
```

本示例中的素性检测方法采用的是 Rabin-Miller 检测算法，Prime 类采用了继承 HugeInt 类的方法，在实现过程中，也可以将素性检测相关的内容直接添加到 HugeInt 类

中，在 HugeInt 类中实现素性检测。

Prime 类中各成员函数的作用如下：

- calcK()——用于计算 Rabin-Miller 素性检测算法中用到的常量 k。
- calcQ()——用于计算 Rabin-Miller 素性检测算法中用到的常量 q。
- witness()——用于检测相关大数是否是合数的证据。
- millerRabin()——Rabin-Miller 素性检测函数。

calcK()函数的实现

计算 k 的函数主要目的是计算待判断是否为素数的大数在减 1 之后被 2 整除的次数，可以通过判断 n%2 是否为 0 来统计。该函数的详细代码见程序清单 14-24。

程序清单 14-24

```
int Prime::calcK(const HugeInt& hugeInt)
{
    HugeInt zero,one,two;
    HugeInt n=hugeInt;
    int count=0;
    zero.genSmallInt(0);
    one.genSmallInt(1);
    two.genSmallInt(2);
    n=n-one;
    while(n>zero)
    {
        if(n%two==zero)
        {
            count++;
            n=n/two;
        }
        else
        {
            break;
        }
    }
    return count;
}
```

calcK()函数的参数为待检测的大数，函数的返回值为大数在减 1 之后被 2 整除的次数，函数的参数实际上是大奇数。函数中声明的 zero，one，two 均为大数，方便用于计算或计算过程中的比较。同时在统计之前需将大数减 1(n＝n－one)。

calcQ()函数的实现

calcQ()函数实际上是要计算 $n-1=2^k q$ 中的 q，函数的参数为大数 HugeInt，函数的返回类型也是大数 HugeInt。calcQ()实现的详细代码见程序清单 14-25。

程序清单 14-25

```
HugeInt Prime::calcQ(const HugeInt& hugeInt)
```

```
02 {
03     HugeInt q,one;
04     one.genSmallInt(1);
05     int tempQ;
06     tempQ=calcK(hugeInt);
07     tempQ=(2<<(tempQ-1));
08     q.genSmallInt(tempQ);
09     return (hugeInt-one)/q;
10 }
```

在 q 的计算过程中 2^k 的计算直接通过移位运算来完成，具体见代码行第 7 行，在计算过程中与 calcK()函数相似，也用到了大数减 1 运算，也需要定义大数的“1”来实现大数的减 1 运算。

witness()函数的实现

witness()函数是针对有个随机数来判断待测大数是否是合数，函数的具体代码见程序清单 14-26。

程序清单 14-26

```
bool Prime::witness(const HugeInt& a,const HugeInt &n)
{
    int k1=calcK(n);
    HugeInt q1=calcQ(n);
    HugeInt one;
    one.genSmallInt(1);
    int i;
    HugeInt x0,x1;
    x0=expMod(x0,q1,n);
    for(i=1;i<=k1;i++)
    {
        total++;
        HugeInt q2;
        q2.genSmallInt(2<<(i-1));
        q2=q2*q1;
        x1=expMod(a,q2,n);
        if(x1==one && x0!=one && x0!=(n-one))
        {
            return false;
        }
        x0=x1;
    }
    if(x1!=one)
    {
        return true;
    }
    return false;
```

```
}
```

witness()函数包括两个参数，一个是待检测的大数，另一个是用来检测的大数，函数的返回类型是 bool 型。

millerRabin()函数的实现

millerRabin()函数用于检测给定的大数是否是素数，函数的具体代码见程序清单 14-27。

程序清单 14-27

```
bool Prime::millerRabin(HugeInt& n,int s)
{
    HugeInt two,three;
    two.genSmallInt(2);
    three.genSmallInt(3);
    int i;
    int length=n.getLength(n);
    HugeInt a;
    for(i=0;i<s;i++)
    {
        a.genHugeInt(length);
        a=a%(n-three)+two;
        if(witness(a,n))
        {
            return false;
        }
    }
    return true;
}
```

millerRabin()函数的参数有两个，一个是待检测的大数，另一个是检测的次数，函数的返回值是 bool 型。函数中的 a 是随机生成的大数，用于检测大数的素性，其值的范围为 $1<a<n-1$，随机选取 a 的次数由可能是素数的所需的概率来确定，即由函数的参数 s 来确定。

在程序清单 14-27 中，素性检测采用的是简单的 Rabin-Miller 检测算法，在实际检测中还可以通过其他一些方法有效加快素性检测的速度，例如：用小于 1000 的小素数先作为检测的因子，对待检测的大数进行验证，这样可以较快将非素数排除，大大缩短素性检测的时间。比如，如果一个奇数的尾数为 5 或 5 的倍数，那么直接计算 $n\%5=0$，就可以确定该大数不是素数。使用小于 1000 的小素数作为检测的因子，实现的方法也比较简单，只需将这些小素数存放到相应数组，在使用时再将这些小素数放入向量，并应用于相应的检测即可。

素数生成

素数生成可以通过自定义一个相关的函数实现，素数的长度可以通过函数的参数来确定，获得素数的函数的详细代码见程序清单 14-28。

程序清单 14-28

```
HugeInt getPrime(int length)
```

```
{
    HugeInt hugeInt1,zero,one;
    Prime prime;
    zero.genSmallInt(0);
    one.genSmallInt(1);
    hugeInt1.genHugeInt(length);
    if(hugeInt1%two==zero)
    {
        hugeInt1=hugeInt1+one;
    }
    while(1)
    {
        cout<<"hugeInt="<<hugeInt1;
        if(prime.millerRabin(hugeInt1,10))
        {
            break;
        }
        else
        {
            hugeInt1.genHugeInt(length);
            if(hugeInt1%two==zero)
            {
                hugeInt1=hugeInt1+one;
            }
        }
    }
    return hugeInt1;
}
```

获得素数的基本过程是：

(1) 随机生成一个所需长度的大数。

(2) 判断该数是否是奇数，若不是奇数，则将该大数加 1。

(3) 对生成的大数进行素性检测，若该大数是素数则返回该大数。若该大数不是素数，则返回到第一步。

重复上述过程直到生成所需的素数为止。

getPrime()函数既可以作为独立的函数，也可以作为 Prime 类的一个成员函数，在示例中是作为一个独立的函数，而 Prime 类只作为素性检测的类。

14.5.3 RSA 算法的实现

RSA 算法是建立在大数运算和素性检测的基础上，用到了欧几里得运算、乘法逆元计算和幂模运算等，RSA 算法通过 RSA 类来实现，RSA 类的基本结构见图 14-7。

RSA 类的声明见程序清单 14-29。

RSA	
- p	: HugeInt
- q	: HugeInt
- e	: HugeInt
- d	: HugeInt
- n	: HugeInt
- phi	: HugeInt
- M	: HugeInt
- C	: HugeInt
- decM	: HugeInt
+ setPrime (const HugeInt& P, const HugeInt& Q)	: void
+ getKey ()	: void
+ setPlainText (const HugeInt& plainText)	: void
+ encryption ()	: void
+ decryption ()	: void
+ display ()	: void

图 14-7　RSA 类的基本结构

程序清单 14-29

```
class RSA
{
    public:
        void setPrime(const HugeInt& P,const HugeInt& Q);
        void getKey();
        void setPlainText(const HugeInt& plainText);
        void encryption();
        void decryption();
        void display();
    private:
        HugeInt p;
        HugeInt q;
        HugeInt e;
        HugeInt d;
        HugeInt n;
        HugeInt phi;
        HugeInt M;
        HugeInt C;
        HugeInt decM;
};
```

RSA 类中各成员函数和成员变量的作用如下：

- p，q——分别为两个大素数，用于 RSA 算法中的密钥计算。
- e，d——分别为加密密钥和解密密钥。
- n——p 和 q 的乘积，用于公钥(e，n)。
- phi——即 $\varphi(n)$，为 $(p-1)(q-1)$，用于计算私钥。
- M——待加密的明文。

- C——加密后的密文。
- decM——将密文脱密后得到的明文。
- setPrime()——设置 p 和 q 的函数。
- getKey()——计算获得公钥和私钥的函数。
- setPlainText()——用于设置待加密的明文。
- encryption()——加密函数。
- decryption()——解密函数。
- display()——辅助函数,用于显示计算结果。

setPrime()函数的实现

setPrime()函数是用于获取计算密钥所需的两个大素数,函数的参数就是对应的大素数,setPrime()函数的具体代码见程序清单 14-30。

程序清单 14-30

```
void RSA::setPrime(const HugeInt& P,const HugeInt& Q)
{
    p=P;
    q=Q;
}
```

函数的内容比较简单,其功能就是将获得的两个大素数赋给 RSA 类中相应的变量。

getKey()函数的实现

getKey()函数主要用于计算公钥与私钥,是 RSA 算法实现中最主要的环节,通过获得的大素数来计算得到加密密钥与解密密钥,函数的具体代码见程序清单 14-31。

程序清单 14-31

```
void RSA::getKey()
{
    HugeInt one;
    one.genSmallInt(1);
    n=p* q;
    phi=(p-one)* (q-one);
    e.genHugeInt(4);
    while(1)
    {
        if(gcd(e,phi)==one)
        {
            break;
        }
        else
        {
            e.genHugeInt(4);
        }
    }
    ExtEuc eu;
```

```
20        eu=invMod(e,phi);
21        d=eu.s;
22        if(d.flag!=1)
23        {
24            d=phi-eu.s;
25        }
26    }
```

getKey()函数通过大素数 p 和 q 计算获得公钥之一 n，然后计算 φ(n)，再选择私钥 e，并满足 e 与 φ(n)互素，代码行的第 7 行到第 18 行为选择 e 的过程，若得到合适的 e，则根据 e 和 n 计算 d，由于在计算乘法逆元时，乘法逆元可能是负值，代码行的第 22 行到第 25 行为处理负值问题。

setPlainText()函数的实现

setPlainText()用于获取明文，函数的详细代码见程序清单 13-32。

程序清单 14-32

```
void RSA::setPlainText(const HugeInt& plainText)
{
    M=plainText;
}
```

setPlainText()函数的参数为大数形式的明文，函数的功能是将需加密的明文传递给 RSA 的成员变量 M。

encryption()函数的实现

encryption()函数功能是计算 $M^e \bmod n$，本质就是进行幂模运算，实现的方法是直接调用大数类中的幂模运算函数，encryption()函数的详细代码见程序清单 14-33。

程序清单 14-33

```
void RSA::encryption()
{
    C=expMod(M,e,n);
}
```

decryption()函数的实现

decryption()函数的功能是计算 $C^d \bmod n$，其计算过程的本质与 encryption()相同，也是进行幂模运算，decryption()函数的详细代码见程序清单 14-34。

程序清单 14-34

```
void RSA::decryption()
{
    decM=expMod(C,d,n);
}
```

display()函数的实现

display()函数主要是方便检验计算结果，函数的详细代码见程序清单 14-35。

程序清单 14-35

```
void RSA::display()
{
    cout<<"p="<<p;
    cout<<"q="<<q;
    cout<<"n="<<n;
    cout<<"phi="<<phi;
    cout<<"e="<<e;
    cout<<"d="<<d;
    cout<<"M="<<M;
    cout<<"C="<<C;
    cout<<"decM="<<decM;
}
```

display()函数直接显示各类大数,该功能使用了大数类中的输出运算符的重载。

14.5.4 RSA 加密算法测试

RSA 加密算法的测试可以通过主函数直接进行测试,或编写一个相应的测试函数用于测试,在本例中直接使用主函数进行测试,测试代码见程序清单 14-36。

程序清单 14-36

```
01 int main()
02 {
03     RSA rsa;
04     HugeInt P,Q,M;
05     M.genHugeInt(5);
06     rsa.setPlainText(M);
07     P=getPrime(8);
08     Q=getPrime(8);
09     rsa.setPrime(P,Q);
10     rsa.getKey();
11     rsa.encryption();
12     rsa.decryption();
13     rsa.display();
14     return 0;
15 }
```

测试过程中的明文通过随机方法生成,在实际应用中可以通过文件读入明文,并转换为 16 进制的格式即可。在代码行的第 7 行和第 8 行为获取两个大素数,大素数的长度可以根据需要来确定,示例中的参数为 8,则大素数的长度为 $8\times32=256$,密钥的长度为 512。在获得所需的大素数之后就计算公钥和私钥,在利用公钥和私钥分别进行加密和解密测试。测试得到的结果如下:

```
p=5404365625b0597b5e1d6c631f6465167d455314117129ba0096266f277a0ac1
```

```
q= 41a465d51bab4d3c0f7f469232d534212b85746d43ad555561d161d839720e69
n= 158b05ea48c7c821f0e55357502e4e8a124e999156fdc6ae435ef529aadf3db8
   31878b3703858cce8c28e8f079145df8005582a7a121a647c696cc55a096f729
phi= 158b05ea48c7c821f0e55357502e4e8a124e999156fdc6ae435ef529aadf3db7
     9bdeef0bc229e6171e8c35fb26dac4c0578abb264c032738642f440e3faade00
e= 1916183833ce1c5331c16c9c291c0eb7
d= 54dec16a8f8bf338b79a50d13d704b3e0ede77115fbb5df4b234ae29514e2ee
   d155614b4d3c489281a28dfd7380edc2a1816e4da6c78b36206726dd07337707
M= 6d8a6435467c71ed615540fe67a028d52ee805d0
C= 140175c657d783fe516ea901f3a5941dc5a70224f185c43f8480ed72db12a7c2
   17ceaec24004e809e33e5a6430f78458f1254b57655510ba625e81b146a15a95
decM= 6d8a6435467c71ed615540fe67a028d52ee805d0
```

最后得到的结果为 M＝decM。

在实际应用中密钥通常在计算得到后需要保存，一般可以使用文件的形式进行保存，在使用时再读取密钥，同样，明文也是从文件中获取。

若需要使用 1024 位密钥进行加密和解密，只需要将获取大素数的函数的参数做相应的修改，即 P＝getPrime(16)，Q＝getPrime(16)。当然，也通过变量的形式来设置密钥的长度。

14.6 习题与实践题

14.6.1 习题

1. 计算 3^{1000} mod 67。
2. 计算 5 模 13 的乘法逆元。
3. 使用 Rabin-Miller 素性检测法判断 23 是否是素数。
4. 使用 Solovag-Strassen 素性检测法判断 17 是否是素数。
5. 使用 Lehmann 素性检测法判断 30 是否是素数。
6. 使用 AKS 素性检测法判断 21 是否是素数。
7. 简要说明大数除法的基本原理，并根据大数除法的基本原理计算 45/9，给出详细计算过程。
8. 假设 RSA 算法中的 $p=17$，$q=23$，$M=34$，试给出计算示例说明 RSA 加密和解密的基本过程，其中公钥 e 根据相关条件自己进行选择。

14.6.2 实践题

1. 参考 14.5 节的相关内容，编程实现 Solovag-Strassen 素性检测法。
2. 参考 14.5 节的相关内容，编程实现 RSA 加密算法，要求大数的处理方式采用数组的形式进行处理。
3. **百万富翁问题** 有两个百万富翁在街头相遇，他们想比较谁更富有(即谁的财富更多)，但又不想让对方了解自己的财富有多少？如果他们能找到一个双方都可信的第三方来

做这件事，则问题很容易就解决。但如果其中一个百万富翁除了自己以外，谁都不相信，这样问题就比较困难了，那么如何在不需要借助第三方的情况下比较他们财富的多少？

这个问题是由华裔计算机学家姚启智在 1982 年提出的，这个问题又被称为姚氏百万富翁问题。

假设富翁 A 和富翁 B 的财富值分别为 a 和 b，并且财富值的大小范围为：$1 \leqslant a, b \leqslant N$，其中 N 是一个正整数，他们的问题可以通过一个公钥算法来完成。假设他们选择了 RSA 公钥算法，并且他们两人都是诚实的，那么，他们的问题可以通过以下步骤完成：

(1) 富翁 B 产生他的公私钥对(KU_B, KR_B)。

(2) 富翁 A 随机选择一个较大的正整数 x，计算

$$c = E_{KU_B}(x)$$

(3) 富翁 A 计算 $c' = c - a$，并将 c' 发送给富翁 B。

(4) 富翁 B 对于 $i = 1, 2, \cdots, N$ 计算

$$y_i = D_{KR_B}(c' + i)$$

(5) 富翁 B 随机选择一个大于 N 的素数 p，对于 $i = 1, 2, \cdots, N$ 计算：

$$z_i = y_i \bmod p$$

(6) 对于每个 i 检验 $0 < z_i < p - 1$ 是否成立？且对所有 $i \neq j$ 检验 $|z_i - z_j| \geqslant 2$ 是否成立？如果有一不成立则返回上一步。

(7) 富翁 B 将下面序列发送给富翁 A：

$$z_1, z_2, \cdots, z_b, z_{b+1} + 1, z_{b+2} + 1, \cdots, z_N + 1, p$$

(8) 富翁 A 检查该序列中的第 a 个数是否是关于模 p 与 x 同余？若同余则 $a \leqslant b$，否则 $a > b$。

计算过程示例：

假设富翁 A 的财富 $a = 7$，富翁 B 的财富 $b = 9$，双方设定的财富上限值 $N = 12$。

(1) 富翁 A 与富翁 B 共同选择 RSA 算法的 p 为 11，q 为 17，计算得到 RSA 算法的 $n = 187$。

(2) 富翁 B 计算公钥和私钥，计算 $\varphi(n) = (p-1)(q-1) = (11-1) \times (17-1) = 160$，并选择 $KU_B = 7$ 作为公钥(7 与 160 互素)，并计算得到私钥 $KR_B = 23$，并公布公钥。

(3) 富翁 A 随机选择整数 $x = 16$，并计算

$$c = E_{KU_B}(x) = 16^7 \bmod 187 = 135$$

并计算 $c' = c - a = 135 - 7 = 128$，并将 128 发送给富翁 B。

(4) 富翁 B 对于 $i = 1, 2, \cdots, 12$ 计算

$$y_i = D_{KR_B}(c' + i) = D_{KR_B}(128 + i) = (128 + i)^{23} \bmod 187$$

计算结果如下：

i	1	2	3	4	5	6	7	8	9	10	11	12
y_i	39	3	109	55	23	8	16	119	103	128	79	149

(5) 富翁 B 随机选择一个大于 N 的素数 $p = 23$，对于 $i = 1, 2, \cdots, 12$ 计算：$z_i = y_i \bmod p$，计算结果如下：

i	1	2	3	4	5	6	7	8	9	10	11	12
z_i	16	3	17	9	0	8	16	4	11	13	10	11

根据第 6 条检测步骤，$p=23$ 不符合要求，重新选择 $p=83$，计算结果如下：

i	1	2	3	4	5	6	7	8	9	10	11	12
z_i	39	3	26	55	23	8	16	36	20	45	79	66

当 $p=83$ 时符合第 6 条检测步骤，则富翁 B 将以下序列发送给富翁 A：

$$39 \quad 3 \quad 26 \quad 55 \quad 23 \quad 8 \quad 16 \quad 36 \quad 20 \quad 45+1 \quad 79+1 \quad 66+1 \quad 83$$

注意，在第 b 个数之后，每个数加 1，最后一个数为 p，不用加。

(6) 富翁 A 检测第 7 个数是 16，它是与 $x=16$ 关于模 83 同余，所以富翁 A 的结论是：$a\leqslant b$。富翁 A 得出结论后将结论告诉富翁 B。

程序要求 编写一个程序帮助这两个富翁解决财富比较问题，并假设他们的财富比较问题在整型数据范围内可以得到解决，使用的公钥算法为 RSA 算法。

4. **平均薪水问题** 假设某公司有 n 个职员，他们想了解这 n 个职员的平均薪水是多少？但是，每个职员又不想让其他职员知道自己的薪水是多少，那么在这种情况下他们如何计算这些职员的平均薪水呢？

假设公司的 n 个职员分别是 $A_1, A_2, \cdots, A_n$，他们的薪水分别是 $a_1, a_2, \cdots, a_n$，这 n 个职员在不泄露自己薪水的情况下也可以计算得到 n 个职员的平均薪水。假设所有的职员都是诚实的，那么平均薪水的具体计算过程如下：

(1) 所有公司职员共同确定一个公钥算法，例如，使用 RSA 算法，然后每个职员确定各自的公私钥对(KU_i, KR_i)。

(2) 职员 A_1 随机选择一个数 x 并加上他的薪水 $x+a_1$，用 A_2 的公钥进行加密，并利用下式加密：

$$E_{KU_2}(x+a_1)$$

在加密完成后将加密结果发送给 A_2。

(3) 职员 A_2 使用自己的私钥 KR_2 进行解密，解密公式如下：

$$D_{KR_2}(E_{KU_2}(x+a_1))$$

解密后得到 $x+a_1$，然后再加上自己的薪水 a_2 得到 $x+a_1+a_2$，再使用 A_3 的公钥进行加密，加密的公式如下：

$$E_{KU_3}(x+a_1+a_2)$$

在加密完成后将加密结果发送给 A_3。

(4) A_3 又进行与 A_2 相同的操作，直至到 A_{n-1}。

(5) A_n 用他的私钥进行 KR_n 解密，解密公式如下：

$$D_{KR_n}(E_{KU_n}(x+a_1+\cdots+a_{n-1}))$$

在解密后再加上他的薪水 $x+a_1+a_2+\cdots+a_n$，再使用 A_1 的公钥进行加密，加密公式如下：

$$E_{KU_1}(x+a_1+\cdots+a_{n-1}+a_n)$$

在加密完成后将加密结果发送给 A_1。

(6) A_1 在获得结果后用私钥进行解密，解密公式如下：

$$D_{KR_1}(E_{KU_1}(x+a_1+\cdots+a_{n-1}+a_n))$$

将解密后的结果减去 x，再除以人数便得到平均薪水。A_1 在计算得到平均薪水之后便可以向全体职员公布平均薪水。

计算过程示例

假设职员有 3 个人，分别为 A_1，A_2 和 A_3，他们的薪水为 $a_1=19$，$a_2=18$，$a_3=23$。三个职员共同商定选择 RSA 公钥算法来计算平均工资，并选择的素数 $p=11$，$q=17$。计算得到 $n=187$，由 p 和 q 计算得到 $\varphi(n)=(p-1)(q-1)=(11-1)\times(17-1)=160$，并分别选择各自的公钥及计算得到的私钥如下：

$$A_1 \quad e=7 \quad d=103$$

$$A_2 \quad e=17 \quad d=113$$

$$A_3 \quad e=29 \quad d=29$$

平均工资的具体计算过程如下：

(1) 职工 A_1 选择一个 $x=31$，并与自己的工资相加得到：$31+19=50$，然后利用 A_2 的公钥进行加密，计算 $50^{17} \bmod 187=118$，将计算结果发送给职工 A_2。

(2) 职工 A_2 在收到计算结果后，利用自己的私钥对由 A_1 发送过来的数据进行解密，计算得到：$118^{113} \bmod 187=50$，然后将计算结果加上自己的工资得到：$50+18=68$，再利用职工 A_3 的公钥进行加密，计算 $68^{29} \bmod 187=17$，并将计算结果发送给 A_3。

(3) 职工 A_3 在收到计算结果后，利用自己的私钥对由 A_2 传送过来的数据进行解密，计算得到：$17^{29} \bmod 187=68$，然后将计算结果加上自己的工资得到：$68+23=91$，再利用职工 A_1 的公钥进行加密，计算 $91^7 \bmod 187==31$，并将计算结果发送给 A_1。

(4) A_1 在收到计算结果后，使用自己的私钥进行加密。计算得到：$31^{103} \bmod 187=91$，在减去自己选择的 x，得：$91-31=60$，并计算 $60/3=20$，得到平均工资，并将计算结果公布。

程序要求 编写一个程序能够帮助公司职员完成平均薪水计算的任务，并假设这个计算过程可以在整数范围内完成。

第 15 章

Diffie-Hellman 密钥交换算法

Diffie-Hellman 密钥交换算法实际上是一种安全协议，这种协议可以使双方在一个不安全的信道中创建一个密钥，这个密钥可以在后续的通信中作为对称密钥来加密通信内容。Diffie-Hellman 首先由 Whitfield Diffie 和 Martin Hellman 于 1976 年发布，后来发现这个算法已由英国信号情报部门的 Malcolm J. Williamson 设计使用，当时该项设计被列为机密。

15.1 Diffie-Hellman 算法原理

Diffie-Hellman 密钥交换算法是一个匿名的密钥交换协议，是许多认证协议的基础，并且是第一个应用于在非保护信道创建共享密钥的方法。

15.1.1 Diffie-Hellman 密钥交换算法基础

在 Diffie-Hellman 密钥交换算法中，首先通信双方需商定一个素数 p 和素数 p 的本原根(或本原元)g。素数 p 的本原根是一个整数 g，这个整数 g 的幂模 p 可以产生 1 到 $p-1$ 之间的所有整数，即

$$g \bmod p, g^2 \bmod p, \cdots, g^{p-1} \bmod p$$

各不相同，其结果是 1 到 $p-1$ 的一个置换。

示例 15-1 设素数 $p=7$，试判断 2 是否是素数 p 的本原根。

解 由 $p=7$ 计算：

$$2^1 = 2 \equiv 2 \bmod 7,\quad 2^2 = 4 \equiv 4 \bmod 7,\quad 2^3 = 8 \equiv 1 \bmod 7$$

$$2^4 = 16 \equiv 2 \bmod 7,\quad 2^5 = 32 \equiv 4 \bmod 7,\quad 2^6 = 64 \equiv 1 \bmod 7$$

因此，2 不是素数 7 的本原根。

示例 15-2 设素数 $p=7$，试判断 3 是否是素数 p 的本原根。

解 由 $p=7$ 计算：

$$3^1 = 3 \equiv 3 \bmod 7,\quad 3^2 = 9 \equiv 2 \bmod 7,\quad 3^3 = 27 \equiv 6 \bmod 7$$

$$3^4 = 81 \equiv 4 \bmod 7,\quad 3^5 = 243 \equiv 5 \bmod 7,\quad 3^6 = 729 \equiv 1 \bmod 7$$

因此，3 是素数 7 的本原根。

一个素数可能有一个以上的本原根，例如素数 19 的本原根有 2，3，10，13，14，15。

本原根的查找是 Diffie-Hellman 密钥交换算法的一个重要组成部分，测试一个整数 g 是否是素数 p 的本原根可以采用如下方法：

(1) 计算所有 $p-1$ 的素因子 $q_1,q_2,\cdots,q_n$。

(2) 对所有素因子 $q_1,q_2,\cdots,q_n$ 计算：

$$g^{(p-1)/q} \bmod p$$

(3) 如果所有的计算结果都不为 1，则 g 为素数 p 的本原根，如果某个 q 的计算结果为 1，则 g 不是素数 p 的本原根。

示例 15-3 设素数 $p=13$，$p-1=12$ 的素因子是 2 和 3，测试 2 和 3 是否是一个本原根。

解 $2^{(13-1)/2} \bmod 13=12$， $2^{(13-1)/3} \bmod 13=3$

计算结果没有 1，因此 2 是素数 13 的本原根。

同样计算：

$$3^{(13-1)/2} \bmod 13 = 1, \qquad 3^{(13-1)/3} \bmod 13 = 3$$

由于计算结果中有 1，因此 3 不是素数 13 的本原根。

在判断 g 是否是素数 p 的本原根的过程中，需要找到 $p-1$ 所有可能的素因子，寻找 $p-1$ 所有可能的素因子可以采用最简单的寻找方法，即从 2 到 $\sqrt{p-1}$ 逐个测试每个数是否是素数，这种测试方法的测试效率比较低，比较好的寻找方法是埃拉托色尼筛选法(Sieve of Eratosthenes)。

埃拉托色尼筛选法筛选素数的过程为：

(1) 创建待测整数序列，待测整数序列为 $2,3,4,\cdots,n$。

(2) 设第一个数为 q 且等于 2，这个数为素数。

(3) 从 q 开始，每次递增 q，并做标记，这个递增序列为 $q,2q,3q,\cdots$，而 q 本身不做标记。

(4) 寻找下一个大于 q 且未作标记的数，并重新记作 q，重复第(3)步的操作，直到所有待测数处理完毕。

(5) 所有未作标记的数都是素数。

在获得所有可能的素因子之后，通过判断是否是 $p-1$ 的因子就可获得所有 $p-1$ 的素因子。

15.1.2 Diffie-Hellman 密钥交换算法计算过程

假设通信双方分别为 A 和 B，那么，Diffie-Hellman 算法首先由 A 和 B 双方协商一个大素数 p 和模 p 的本原根 g，这两个整数并不一定要求是秘密的，可以通过不安全的信道进行协商。然后，通过下列步骤来计算密钥：

(1) A 方首先选择一个大的随机整数 x，$1\leqslant x\leqslant p-2$，计算 $X=g^x \bmod p$，并将 X 发送给 B 方。

(2) B 方也选择一个大的随机整数 y，$1\leqslant y\leqslant p-2$，计算 $Y=g^y \bmod p$，并将 Y 发送给 A 方。

(3) A 方计算 $K_a=Y^x \bmod p$，作为密钥。

(4) B 方计算 $K_b=X^y \bmod p$，作为密钥。

因为 $Y^x=(g^y)^x=g^{xy}=(g^x)^y=X^y$，所以，A 方和 B 方得到的密钥是相同的。由于双方选择的 x 和 y 是保密，如果攻击方想获得密钥 K，它就面临去解离散对数 $X=g^x \bmod p$ 或 $Y=g^y \bmod p$ 的难题。

A,B 双方密钥生成通信的基本模式如图 15-1 所示。

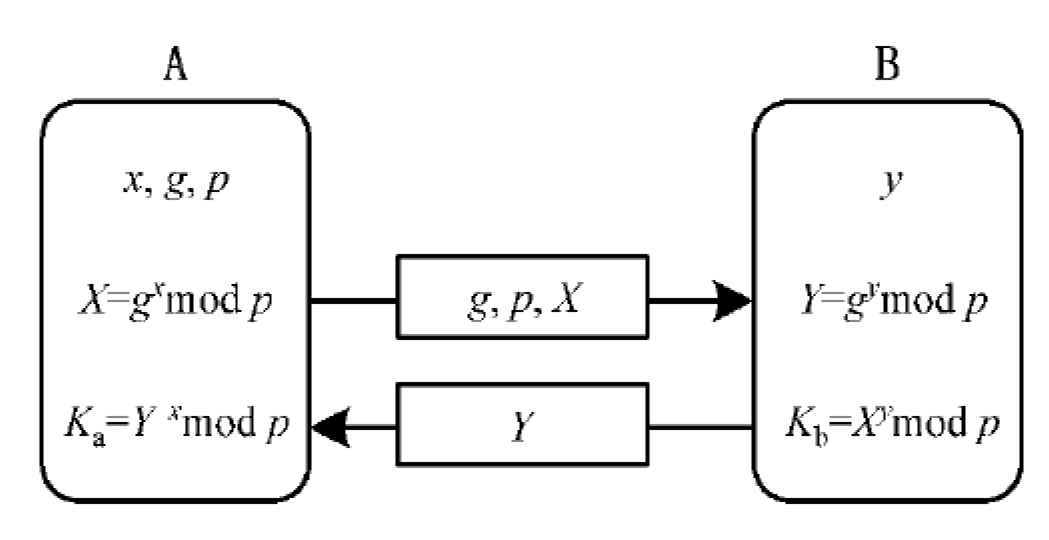


图 15-1 Diffie-Hellman 密钥交换基本模式

示例 15-4 设双方选定素数 $p=23$,并选择本原根 $g=11$,通信中的 A 方选择的整数为 $x=8$,B 方选择的整数为 $y=12$,试计算双方的共享密钥。

解 (1) A 方计算 $X=g^x \bmod p=11^8 \bmod 23=8$,并将 X 发送给 B。

(2) B 方计算 $Y=g^y \bmod p=11^{12} \bmod 23=12$,并将 Y 发送给 A。

(3) A 方计算 $K_a=Y^x \bmod p=12^8 \bmod 23=8$,得到共享密钥。

(4) B 方计算 $K_b=X^y \bmod p=8^{12} \bmod 23=8$,得到共享密钥,并有 $K_a=K_b$。

Diffie-Hellman 算法同样可以用在三方或三方以上的密钥协商,假设通信的三方分别为 A,B,C,那么,A,B,C 三方的共享密钥的生成过程如下:

(1) 通信三方首先商定一个大素数 p 和素数 p 的本原根 g 作为公共参数。

(2) A 方选择一个大的随机整数 x,$1\leqslant x\leqslant p-2$,计算 $X=g^x \bmod p$,并将 X 发送给 B 方。

(3) B 方选择一个大的随机整数 y,$1\leqslant y\leqslant p-2$,计算 $Y=g^y \bmod p$,并将 Y 发送给 C 方。

(4) C 方选择一个大的随机整数 z,$1\leqslant z\leqslant p-2$,计算 $Z=g^z \bmod p$,并将 Z 发送给 A 方。

(5) A 方计算 $Z'=Z^x \bmod p$,并将 Z' 发送给 B 方。

(6) B 方计算 $X'=X^y \bmod p$,并将 X' 发送给 C 方。

(7) C 方计算 $Y'=Y^z \bmod p$,并将 Y' 发送给 A 方。

(8) A 方计算 $K_a=Y'^x \bmod p$。

(9) B 方计算 $K_b=Z'^y \bmod p$。

(10) C 方计算 $K_c=X'^z \bmod p$。

A,B,C 三方密钥生成通信的基本模式如图 15-2 所示。

示例 15-5 设三方选定素数 $p=23$,并选择本原根 $g=11$,通信中的 A 方选择的整数为 $x=10$,B 方选择的整数为 $y=13$,C 方选择的整数为 $z=17$,试计算三方的共享密钥。

解 (1) A 计算 $X=g^x \bmod p=11^{10} \bmod 23=2$,并将 X 发送给 B。

(2) B 方计算 $Y=g^y \bmod p=11^{13} \bmod 23=17$,并将 Y 发送给 C。

(3) C 方计算 $Z=g^z \bmod p=11^{17} \bmod 23=14$,并将 Z 发送给 A。

(4) A 方计算 $Z'=Z^x \bmod p=14^{10} \bmod 23=18$,并将 Z' 发送给 B。

(5) B 方计算 $X'=X^y \bmod p=2^{13} \bmod 23=4$,并将 X' 发送给 C。

(6) C 方计算 $Y'=Y^z \bmod p=17^{17} \bmod 23=11$,并将 Y' 发送给 A。

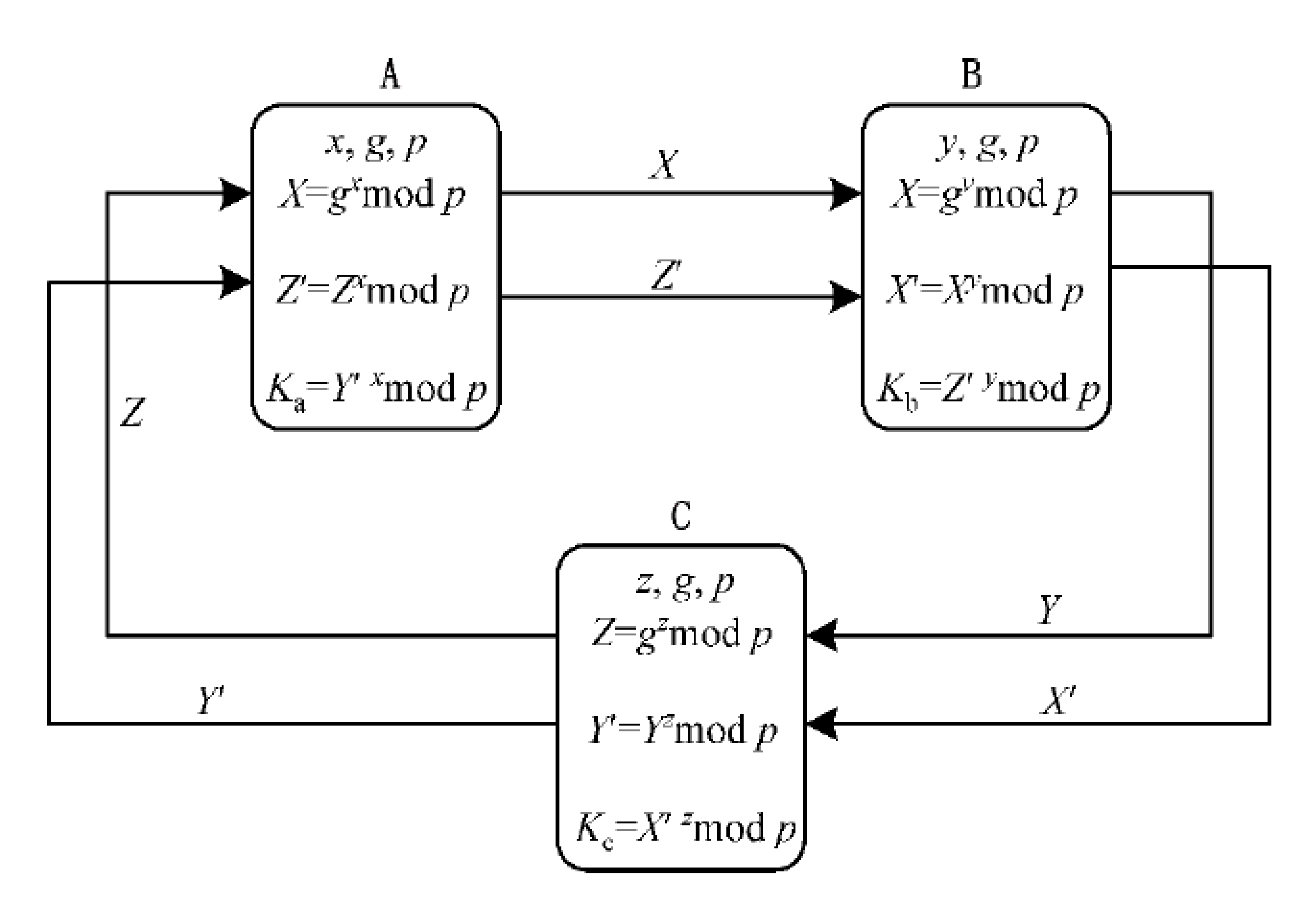


图 15-2 三方密钥生成通信模式

(7) A 方计算 $K_a = Y'^x \bmod p = 11^{10} \bmod 23 = 2$。

(8) B 方计算 $K_b = Z'^y \bmod p = 18^{13} \bmod 23 = 2$。

(9) C 方计算 $K_c = X'^z \bmod p = 4^{17} \bmod 23 = 2$。

多人 Diffie-Hellman 密钥交换模式和上述过程类似，仅增加了计算的轮数。

15.2 Diffie-Hellman 算法实现

Diffie-Hellman 算法通过 Diffie_Hellman 类来实现，Diffie_Hellman 类主要完成生成素因子列表、生成本原根、计算交换数据、计算密钥等，同时还包括一些辅助的功能，如：生成素数、幂模运算等。在本示例中的实现过程主要为了解释 Diffie-Hellman 算法的实现方法，具体使用只需进行一定改造即可。Diffie_Hellman 类的基本组成见图 15-3。

在 Diffie_Hellman 类中重定义了两种数据类型如下：

```
typedef unsigned long long int word64;
typedef unsigned int word32;
```

Diffie_Hellman 类的声明见程序清单 15-1。

程序清单 15-1

```
class Diffie_Hellman
{
    public:
        bool isPrime(word32 n);
        void getPrimeList();
        void genPrime();
        void getBase();
        void display();
        void setX();
        void setY();
        void setG();
```

Diffie_Hellman	
- vPrime	: vector<word32>
- vBase	: vector<word32>
- p	: word32
- x	: word32
- y	: word32
- A	: word32
- B	: word32
- KeyA	: word32
- KeyB	: word32
- g	: word32
+ isPrime (word32 n)	: bool
+ getPrimeList ()	: void
+ genPrime ()	: void
+ getBase ()	: void
+ display ()	: void
+ setX ()	: void
+ setY ()	: void
+ setG ()	: void
+ calcA ()	: void
+ calcB ()	: void
+ calcKeyA ()	: void
+ calcKeyB ()	: void
+ expMod (word32 x, word32 y, word32 p)	: word32

图 15-3　Diffie_Hellman 类的基本组成

```
        void calcA();
        void calcB();
        void calcKeyA();
        void calcKeyB();
        word32 expMod(word32 x,word32 y,word32 p);
    private:
        vector< word32> vPrime;
        vector< word32> vBase;
        word32 p;
        word32 x,y;
        word32 A,B;
        word32 KeyA,KeyB;
        word32 g;
};
```

在 Diffie_Hellman 类中各成员变量的作用如下：

- vPrime——数据类型为向量，用于存储 p－1 的素因子。
- vBase——数据类型为向量，用于存储素数 p 的本原根。
- p——用于计算密钥用的素数。
- x，y——通信双方各自选取的整数，1≤x，y≤p－2。
- A，B——通信双方通过 g，x，y 和 p 计算获得的变量，发送给对方，用于计算密钥。
- KeyA，KeyB——通信双方计算获得的密钥，KeyA 和 KeyB 结果应相同。

- g——用于计算密钥的本原根。

在 Diffie_Hellman 类中各成员函数的作用如下：

- isPrime()——函数参数为 word32 型数据，函数作用为判断函数参数是否为素数。
- getPrimeList()——获得 p－1 的素因子列表。
- genPrime()——用于生成素数 p。
- getBase()——通过素因子列表获得 p 的本原根。
- display()——用于显示计算结果。
- setX()——用于设置通信 A 方的秘密整数。
- setY()——用于设置通信 B 方的秘密整数。
- setG()——用于设置本原根。
- calcA()——用于计算 A 方传递给 B 方的参数。
- calcB()——用于计算 B 方传递给 A 方的参数。
- calcKeyA()——用于计算 A 方的密钥。
- calcKeyB()——用于计算 B 方的密钥。
- expMod()——幂模运算。

示例用于计算 32 位的通信密钥，计算过程包括：生成素数 p、生成本原根 g 和计算双方通信密钥等几部分组成。

15.2.1 生成素数 *p*

素数 *p* 的生成通过函数 genPrime()来完成，其实现的过程是通过生成 32 位随机数，然后判断其是否为素数，若不是素数，则重新生成 32 位随机数，直到获得所需的随机数。函数的具体代码见程序清单 15-2。

程序清单 15-2

```
void Diffie_Hellman::genPrime()
{
    srand(time(0));
    while(1)
    {
        p=rand()&0xFFFF;
        p<<=16;
        p|=rand()&0xFFFF;
        if(isPrime(p))
        {
            break;
        }
    }
}
```

32 位随机数的生成方法是：先生成 16 位的随机数，然后左移 16 位，再与后生成的 16 位随机数进行“或”运算，得到 32 位的随机数。在得到 32 位随机数之后，判断生成的随机数是否为素数，若不是，则重新生成素数，直到得到所需的素数结束。在生成素数的过程

中用到素性判断函数 isPrime(),素数检测的方法可以采用 RSA 算法中的 millerRabin 算法,具体实现方法可以参考程序清单 14-27。

15.2.2　本原根的生成

本原根的生成通过两步完成,第一步是获得 p-1 的所有的素因子,第二步是获得所有的本原根。获取所有素因子是通过函数 getPrimeList()函数来完成,函数 getPrimeList()的详细代码见程序清单 15-3。

程序清单 15-3

```
01 void Diffie_Hellman::getPrimeList()
02 {
03     vPrime.clear();
04     vPrime.push_back(2);
05     word32 temp=p-1;
06     word32 i;
07     for(i=2;i<p;i++)
08     {
09         while(temp%i==0)
10         {
11             temp=temp/i;
12             if(i>vPrime[vPrime.size()-1])
13             {
14                 vPrime.push_back(i);
15             }
16         }
17         if(temp<i)
18         {
19             break;
20         }
21     }
22 }
```

素因子生成的方法是以埃拉托色尼筛选法为基础进行改造,将获得的素因子存储到向量 vPrime。为了方便存储素因子,首先将 2 存放到向量 vPrime 中,为避免将重复的素因子存放到向量 vPrime,在处理素因子的过程中,当找到一个素因子就与向量 vPrime 中的最后一个元素比较,若大于向量 vPrime 中的最后一个元素,就将该素因子存储到向量 vPrime 中,否则就不处理,具体见代码行的第 11 行到第 16 行代码。同时,为避免多余的运算,当变量用于查找素因子的临时变量 temp 小于可能的素因子 i 时查找过程结束。

在获取所有的素因子之后,就需要在素因子中找到本原根,判断查找本原根的任务由函数 getBase()来完成,函数 getBase()的具体代码见程序清单 15-4。

程序清单 15-4

```
void Diffie_Hellman::getBase()
{
```

```
03        vBase.clear();
04        word32 i,j;
05        bool bBase=true;
06        for(i=0;i<vPrime.size();i++)
07        {
08            bBase=true;
09            for(j=0;j<vPrime.size();j++)
10            {
11                if(expMod(vPrime[i],(p-1)/vPrime[j],p)==1)
12                {
13                    bBase=false;
14                    continue;
15                }
16            }
17            if(bBase==true)
18            {
19                vBase.push_back(vPrime[i]);
20            }
21        }
22    }
```

代码行的第 11 行到第 15 行用来判断是否是素数 p 的本原根,若不是素数 p 的本原根就对下一个素因子进行判断,若是就存储到本原根向量 vBase。在本原根的查找过程中用到了幂模运算函数 expMod(),该函数的详细代码见程序清单 15-5。

程序清单 15-5

```
01    word32 Diffie_Hellman::expMod(word32 x,word32 y,word32 p)
02    {
03        word32 result=1;
04        while(y)
05        {
06            if(y&1)
07            {
08                result=(word64(result)*x)%p;
09            }
10            y>>=1;
11            x=(word64(x)*x)%p;
12        }
13        return result;
14    }
```

幂模运算的实现方法与大数运算中的幂模运算实现方法相同,只是由原来的大数改为 32 位的整数,在运算过程中需要注意中间结果的溢出问题,代码行的第 8 行和第 11 行都使用了 word64 来转换中间结果,防止中间结果的溢出。

15.2.3 密钥生成

根据计算所得的本原根 g 和生成的素数 p，通信双方各自选择相应的整数，并计算出密钥所需的参数，发送给对方。本原根根据本原根的计算结果进行选择，本原根的选择通过 setG()函数来完成，setG()函数的详细代码见程序清单 15-6。

程序清单 15-6

```
void Diffie_Hellman::setG()
{
    word32 i;
    cout<<"The generators are:"<<endl;
    for(i=0;i<vBase.size();i++)
    {
        cout<<"("<<i<<") "<<vBase[i]<<endl;
    }
    word32 select;
    cout<<"Input the number of prime factors: ";
    while(cin>>select)
    {
        if(select>=0 && select<vBase.size())
        {
            g=vBase[select];
            break;
        }
        cout<<"Reinput the number: ";
    }
}
```

本原根的选择过程是首先列出备选的本原根，然后根据用户的选择确定具体使用哪个本原根，当选择的本原根的索引出错时，重新进行选择。

在完成本原根的选择之后，就需要确定通信双方各自的秘密数，各自秘密数确定的函数分别为 setX()和 setY()，setX()函数的详细代码见程序清单 15-7。

程序清单 15-7

```
void Diffie_Hellman::setX()
{
    cout<<"Input the secret number of X(2~ "<<(p-2)<<"):";
    while(cin>>x)
    {
        if(x>=2 && x<=(p-2))
        {
            break;
        }
        else
        {
```

```
                cout<< "The number is out of ranger, reinput:";
            }
        }
}
```

通信双方的秘密数的选择需要考虑所选数的范围，若选择超出范围，则重新进行选择，若选择在合理范围内，则选择结束。

setY()函数的实现方法与 setX()的实现方法相同，具体代码见程序清单 15-8。

程序清单 15-8

```
void Diffie_Hellman::setY()
{
    cout<< "Input the secret number of Y(2~ "<< (p-2)<< "):";
    while(cin>> y)
    {
        if(y>=2 && y<=(p-2))
        {
            break;
        }
        else
        {
            cout<< "The number is out of ranger, reinput:";
        }
    }
}
```

setY()函数所需的注意事项与 setX()函数相同。

在完成选择各自秘密数的选择之后，计算出最终计算密钥所需的参数，其计算过程分别由函数 calcA()和函数 calcB()来完成，calcA()函数的具体代码见程序清单 15-9。

程序清单 15-9

```
void Diffie_Hellman::calcA()
{
    A=expMod(g,x,p);
}
```

calcB()函数的具体代码见程序清单 15-10。

程序清单 15-10

```
void Diffie_Hellman::calcB()
{
    B=expMod(g,y,p);
}
```

函数 calcA()和函数 calcB()的实现仅调用了幂模函数，在完成计算密钥所需的参数之后，就可以计算密钥，通信双方的密钥分别通过函数 calcKeyA()和函数 calcKeyB()实现，函

数 calcKeyA()的详细代码见程序清单 15-11。

程序清单 15-11

```
void Diffie_Hellman::calcKeyA()
{
    KeyA=expMod(B,x,p);
}
```

calcKeyB()函数的详细代码见程序清单 15-12。

程序清单 15-12

```
void Diffie_Hellman::calcKeyB()
{
    KeyB=expMod(A,y,p);
}
```

函数 calcKeyA()和函数 calcKeyB()的实现都是调用幂模函数来完成计算，计算获得通信双方所需的密钥，并且，双方的密钥是相同的。

15.2.4 Diffie-Hellman 算法测试

Diffie-Hellman 算法测试通过编写一测试函数来完成，测试函数的具体代码见程序清单 15-13。

程序清单 15-13

```
void test()
{
    Diffie_Hellman diffie;
    diffie.genPrime();
    diffie.getPrimeList();
    diffie.getBase();
    diffie.setG();
    diffie.setX();
    diffie.setY();
    diffie.calcA();
    diffie.calcB();
    diffie.calcKeyA();
    diffie.calcKeyB();
    diffie.display();
}
```

测试函数的实现过程中需要注意类的各个成员函数的调用顺序，在计算完成后通过调用 Diffie_Hellman 类的成员函数 display()来显示各部分的计算结果，函数 display()的详细代码见程序清单 15-14。

程序清单 15-14

```
void Diffie_Hellman::display()
```

```
{
    cout<< "The Prime is: 0x"<< hex<< p<< endl;
    cout<< "The generator is: 0x"<< hex<< g<< endl;
    cout<< "The secret number of A is: 0x"<< hex<< x<< endl;
    cout<< "The secret number of B is: 0x"<< hex<< y<< endl;
    cout<< "The result of Key A is: 0x"<< hex<< KeyA<< endl;
    cout<< "The result of Key B is: 0x"<< hex<< KeyB<< endl;
}
```

具体测试可以通过主函数 main()来驱动,测试结果如下:

```
The generators are:
(0) 3
(1) 23
Input the number of prime factors: 1
Input the secret number of X(2~ 630156989):98546
Input the secret number of Y(2~ 630156989):78524
The Prime is: 0x258f6ebf
The generator is: 0x17
The secret number of A is: 0x180f2
The secret number of B is: 0x132bc
The result of Key A is: 0x1d8d6d60
The result of Key B is: 0x1d8d6d60
```

各测试结果以十六进制的方式进行。

本示例的主要目的是为了解释 Diffie-Hellman 算法的计算方法,具体应用时需进行适当的改变,例如,示例中的素数的选择是通过随机的方法生成,而实际应用则可以在生成所需的素数之后,在计算过程中通过其他方式来读取该素数,另外,密钥计算是在一个程序中完成,而实际应用中是双方计算各自的密钥,再利用计算所得的密钥进行通信,而不是在一起计算密钥。

15.3 习题与实践题

15.3.1 习题

1. 假设素数 $p=13$,试判断 3 是否是 p 的本原根。
2. 简要说明 Diffie-Hellman 密钥交换算法的基本原理。
3. 假设通信双发选择 Diffie-Hellman 算法,并设双方选定素数 $p=23$,并选择本原根 $g=11$,通信中的 A 方选择的整数为 $x=7$,B 方选择的整数为 $y=9$,试计算双方的共享密钥。

15.3.2 实践题

参考 15.2 节的 Diffie-Hellman 密钥交换的算法的实现过程,编程实现 Diffie-Hellman 密钥交换算法。要求:使用数组或指针形式处理相关数据。

第16章 Elgamal 加密算法

Elgamal 算法是由 Taher Elgamal 于 1984 年提出的算法，Elgamal 算法共有两部分组成，Elgamal 加密算法和 Elgamal 数字签名算法，Elgamal 算法的基础是 Diffie-Hellman 算法，其安全性主要依赖于计算有限域上离散对数这一难题。目前该算法主要应用于 PGP 协议，在本章中主要介绍 Elgamal 加密算法。

16.1 Elgamal 加密算法原理

Elgamal 公钥加密算法主要由三部分组成：密钥的生成、加密算法和解密算法。

密钥生成

Elgamal 公钥加密算法的密钥包括公钥和私钥两部分，假设通信中的 A 方负责计算密钥，其他方使用密钥，A 方生成密钥的过程如下：

（1）生成随机大素数 p 以及 p 的本原根 g。

（2）随机选择私钥 x，$1\leqslant x\leqslant p-2$。

（3）计算 $y=g^x \bmod p$。

计算得到的公钥为(p,g,y)，私钥为 x。

加密

假设 B 要与 A 方进行通信，B 所要加密的明文为 m，并使用 Elgamal 加密算法，那么，B 方执行的操作如下：

（1）从 A 方获得加密所需的公钥(p,g,y)。

（2）将明文 m 转换为整型数据，其范围在 $0\sim p-1$ 之间。

（3）选择一随机数 k，$1\leqslant k\leqslant p-2$。

（4）计算 $a=g^k \bmod p$ 和 $b=my^k \bmod p$。

（5）发送密文 $c=(a,b)$给 A。

解密

当 A 接收到从 B 发送的密文之后，需对密文进行解密，解密的过程如下：

（1）使用私钥 x 计算 $a^{p-1-x} \bmod p=a^{-x} \bmod p=g^{-xk} \bmod p$。

（2）通过计算 $m=(a^{-x})b \bmod p$。

上述解密过程的原理为$(a^{-x})b=g^{-xk}my^k=g^{-xk}mg^{x^k}\equiv m(\bmod p)$。

示例 16-1 假设 A,B 双方进行通信,通信过程由 A 方确定密钥,A 方选择的素数 $p=23$,选择的本原根 $g=11$,选择的私钥 $x=9$,B 方加密时选择的 $k=17$,B 方待加密的明文为 19,试给出 A,B 双方进行通信的详细过程。

解 $p=23, g=11, x=7$。

A 方计算密钥:

$$y = g^x \bmod p = 11^9 \bmod 23 = 19$$

得到公钥为$(p,g,y)=(23,11,19)$,私钥为 7,将公钥发送给 B 方,A 方保存私钥,待解密时使用。

B 方利用公钥进行加密,计算:

$$a = g^k \bmod p = 11^{17} \bmod 23 = 14$$

和

$$b = my^k \bmod p = 19 \times 19^{17} \bmod 23 = 8$$

将计算结果 $c=(a,b)=(14,8)$发送给 A 方。

A 方收到 B 方发送的密文 c 后,对密文 c 进行解密还原为明文 m,计算

$$a^{p-1-x} \bmod p = 14^{23-1-9} \bmod p = 11$$

和

$$m = a^{-x}b \bmod p = 11 \times 8 \bmod 23 = 19$$

得到明文 $m=19$。

Elgamal 加密算法主要在两方面有较多应用,第一个应用是直接对明文进行加密,当直接对明文加密时,加密后的密文得到较大扩展,使密文与明文相比成倍扩大。Elgamal 加密算法的第二个应用是用于对称密钥的加密,当通信双方使用对称加密算法进行加密时,为防止攻击方在信道中窃取密码,故采用 Elgamal 加密算法对密钥进行加密,在传递给对方以确保对称密钥的安全。

16.2 Elgamal 加密算法实现

Elgamal 加密算法的实现采用模拟双方进行加密通信的过程进行实现,假设通信双方分别为 A 和 B,由 A 来生成公钥。算法实现共分为三部分,由 A 完成的部分包括密钥的生成和解密,由 B 来完成的部分为加密部分,算法测试由 A,B 两部分共同进行。

16.2.1 密钥的生成与解密的实现

密钥的生成与解密由通信方中的 A 方来完成,在示例中是通过 ElgamalA 类来实现相关的功能,公钥与私钥都在 32 位的范围内,重点在于揭示 Elgamal 算法的基本实现方法,若采用更高位数来实现,只需使用第 14 章的大数运算,并根据 Elgamal 算法具体应用即可。为方便使用,相关的数据类型进行重新定义,具体如下:

```
typedef unsigned int word32;
typedef unsigned long long int word64;
```

word64 型数据主要用于处理计算过程中的中间结果,以防止中间结果的溢出。

ElgamalA 类的基本结构如图 16-1 所示。

ElgamalA	
- cipherTextA	: word32
- cipherTextB	: word32
- decipherText	: word32
- p	: word32
- g	: word32
- x	: word32
- y	: word32
+ ElgamalA ( )	
+ genPrime ( )	: void
+ genBase ( )	: void
+ isPrime (word32 p)	: bool
+ getPrime ( )	: void
+ setPrivateKey ( )	: void
+ expMod (word32 x, word32 y, word32 p)	: word 32
+ setPubKey ( )	: void
+ getPubKey ( )	: void
+ decryption ( )	: void
+ run ( )	: void
+ instruction ( )	: void

图 16-1　ElgamalA 类的基本结构

ElgamalA 类的具体声明见程序清单 16-1。

程序清单 16-1

```
class ElgamalA
{
    public:
        ElgamalA();
        void genPrime();
        void genBase();
        bool isPrime(word32 p);
        void getPrime();
        void setPrivateKey();
        word32 expMod(word32 x,word32 y,word32 p);
        void setPubKey();
        void decryption();
        void run();
        void instruction();
    private:
        word32 cipherTextA,cipherTextB;
        word32 decipherText;
        word32 p;
        word32 g;
        word32 x,y;
};
```

ElgamalA 类中的各成员变量的作用如下：

- cipherTextA,cipherTextB——用于存储 B 方发送过来的两个加密后数据,用于解密。
- decipherText——存储 cipherTextA 和 cipherTextB 解密后的数据。
- p——公钥中的素数。
- g——素数 p 的本原根。
- x——私钥。
- y——通过 g,p,x 计算后获得的公钥。

(p,g,y)为计算后需公布或发送给 B 方的公钥。

ElgamalA 类中的各成员函数的作用如下:

- ElgamalA()——构造函数,用于初始化各成员变量。
- genPrime()——用于生成公钥中的素数 p。
- genBase()——用于计算并选择素数 p 的本原根。
- isPrime()——用于判断相应的参数是否是素数。
- getPrime()——从文件获取已生成的素数。
- setPrivateKey()——设置 A 方的私钥。
- expMod()——幂模运算。
- setPubKey()——输出公钥到文件。
- decryption()——用于解密从 B 方传递过来的密文。
- run()——用于测试运行。
- instruction()——用于提示操作内容。

16.2.1.1 密钥生成

ElgamalA 类中密钥的生成包括两部分:公钥的生成和私钥的生成。公钥和私钥生成的基本过程为:先生成公钥中的素数 p,再生成公钥中素数 p 的本原根 g,然后根据素数 p 来获取私钥 x,最后计算公钥中的最后一个参数 y。

获取素数

素数的获得有两种方法,一种是通过随机数来生成,另一种是直接通过文件来获得已经生成的素数,直接获得素数的函数为 getPrime(),getPrime()函数的详细代码见程序清单 16-2。

程序清单 16-2

```
void ElgamalA::getPrime()
{
    ifstream primeIn("prime.txt");
    primeIn>>hex>>p;
}
```

从文件获得素数的方法是直接从文件流读取已保存的素数,在函数中直接声明了文件流变量,使用这种方法来声明是为了方便理解,也可以通过声明类中的私有变量,并通过构造函数初始化的方法来处理,后续的文件处理均采用在相应函数中直接声明和处理。

如果不采用文件读取素数的方法来获得素数,那么,可以采用直接生成素数的方法来生成所需的素数,完成这一功能的函数为 genPrime(),该函数是通过生成随机数的方法来生

成素数，genPrime()函数与 Diffie-Hellman 中的素数生成方法一样，具体可参考程序清单 15-2 来实现，只需将函数名称改为 void ElgamalA::genPrime()即可。

在素数生成完毕后，同样将素数保存到"prime.txt"的文件，当需要时可以直接从该文件中获得素数。

在素数生成过程中使用到函数 isPrime()来判断随机生成的 32 位数是否是素数，isPrime()函数与 Diffie-Hellman 中 isPrime()函数的实现方法相同，同样可参考 RSA 算法中的程序清单 14-27 来实现。

计算本原根

在获取相应的素数之后需计算素数 p 的本原根 g，本原根 g 的计算通过函数 genBase()来完成，genBase()的详细代码见程序清单 16-3。

程序清单 16-3

```
void ElgamalA::genBase()
{
    vector<word32> vPrimeList;
    vPrimeList.push_back(2);
    word32 temp=p-1;
    word32 i,j;
    for(i=2;i<p;i++)
    {
        while(temp%i==0)
        {
            temp=temp/i;
            if(i>vPrimeList[vPrimeList.size()-1])
            {
                vPrimeList.push_back(i);
            }
            if(temp<i)
            {
                break;
            }
        }
    }
    vector<word32> vBase;
    bool bBase=true;
    for(i=0;i<vPrimeList.size();i++)
    {
        bBase=true;
        for(j=0;j<vPrimeList.size();j++)
        {
            if(expMod(vPrimeList[i],(p-1)/vPrimeList[j],p)==1)
            {
                bBase=false;
                continue;
```

```
33                  }
34              }
35              if(bBase==true)
36              {
37                  vBase.push_back(vPrimeList[i]);
38              }
39          }
40          cout<<"Select g from the list:"<<endl;
41          for(i=0;i<vBase.size();i++)
42          {
43              cout<<i<<". "<<vBase[i]<<endl;
44          }
45          word32 select;
46          cout<<"Input your choice: ";
47          while(cin>>select)
48          {
49              if(select>=0 && select<vBase.size())
50              {
51                  g=vBase[select];
52                  break;
53              }
54              cout<<"Reinput your choice: ";
55          }
56      }
```

本原根通过以下过程获得：

（1）生成素因子列表：代码行的第 7 行到第 21 行为生成 p—1 的素因子列表。

（2）生成本原根列表：代码行的第 24 行到第 39 行为根据素因子列表生成本原根列表。

（3）选择本原根：代码行的第 40 行到第 55 行为用户根据本原根列表选择本原根。

在计算本原根的过程中使用了函数 expMod()进行幂模运算，expMod 函数的实现方法与 Diffie-Hellman 中的幂模运算函数 expMod()完全相同，具体实现可参考 Diffie-Hellman 中的实现方法。

获取私钥

在生成本原根之后可以通过用户输入来获得私钥 x，获取私钥通过 setPrivateKey()函数来完成，setPrivateKey()函数的详细代码见程序清单 16-4。

程序清单 16-4

```
void ElgamalA::setPrivateKey()
{
    while(1)
    {
        cout<<"Input your private key(1~ "<<(p-2)<<"):";
        cin>>x;
        if(x<=1 || x>(p-2))
```

```
        {
            cout<< "Out of range, reinput!"<< endl;
        }
        else
        {
            break;
        }
    }
    ofstream priKeyOut("priKey.txt");
    priKeyOut<< x;
}
```

获取私钥函数个功能为：根据用户输入判断是否在合理的范围内，若在合理的范围内，则输入结束，并将私钥保存到文件“priKey.txt”，否则提示用户重新输入私钥。

公钥参数设置

公钥参数设置通过函数 setPubKey()来实现，函数的详细代码见程序清单 16-5。

程序清单 16-5

```
void ElgamalA::setPubKey()
{
    y= expMod(g,x,p);
    ofstream pubKeyOut("pubKey.txt");
    pubKeyOut<< p<< " "<< g<< " "<< y;
}
```

函数功能为计算公钥的最后一个参数 y，在计算完成后将公钥保存到文件“pubKey.txt”，公钥既用于 B 方加密文件时使用，也用于 A 方在获得文件后解密文件使用。

16.2.1.2 解密算法实现

ElgamalA 类中的另一个主要功能是解密，当从 B 方获取经过加密的密文时，需使用相关的解密函数进行解密，解密函数为 decryption()，其功能为分别对加密后的两个密文进行解密，decryption()函数的具体代码见程序清单 16-6。

程序清单 16-6

```
void ElgamalA::decryption()
{
    ifstream cipherTextIn("cipherText.txt");
    ifstream pubKeyIn("pubKey.txt");
    ifstream priKeyIn("priKey.txt");
    cipherTextIn>> cipherTextA>> cipherTextB;
    pubKeyIn>> p>> g>> y;
    priKeyIn>> x;
    word32 temp= expMod(cipherTextA, (p- 1- x),p);
    decipherText= (word64(temp) * cipherTextB)%p;
    cout<< "cipherTextA= "<< cipherTextA<< endl;
```

```
12         cout<< "cipherTextB= "<< cipherTextB<< endl;
13         cout<< "decipherText= "<< decipherText<< endl;
14    }
```

在解密过程中，首先从相关的文件中读取公钥、私钥和密文，密文存放在“cipherText.txt”文件中，各文件流声明见第 3 行到第 5 行代码，然后利用解密方法对密文进行解密，获得还原后的明文。

16.2.2 加密的实现

加密过程主要由 B 方来完成，在算法实现中是通过 ElgamalB 类来完成相应的功能，ElgamalB 类的基本结构如图 16-2 所示。

ElgamalB	
- m : word32	
- a : word32	
- b : word32	
- p : word32	
- g : word32	
- y : word32	
- k : word32	
+ gcd (int n, int k)	: int
+ expMod (word32 x, word32 y, word32 p)	: word32
+ getPubKey ()	: void
+ getK ()	: void
+ getPlainText ()	: void
+ encryption ()	: void
+ run ()	: void

图 16-2 ElgamalB 类的基本结构

ElgamalB 类的数据处理方法与 ElgamalA 类的数据处理方法相同，同样使用 word32 和 word64 两种数据类型来处理数据，ElgamalB 类的具体声明见程序清单 16-7。

程序清单 16-7

```
class ElgamalB
{
    public:
        int gcd(int n,int k);
        word32 expMod(word32 x,word32 y,word32 p);
        void getPubKey();
        void getK();
        void getPlainText();
        void encryption();
        void run();
    private:
        word32 m;
        word32 a,b;
        word32 p,g,y;
```

```
        word32 k;
};
```

ElgamalB 类中的各成员变量的作用如下：

- m——用于存储明文数据。
- a，b——用于存储对明文 m 进行加密后的数据。
- p，g，y——加密算法中的公钥。
- k——加密时选用的参数。

ElgamalB 类中的各成员函数的作用如下：

- gcd()——欧几里得函数。
- expMod()——幂模运算函数。
- getPubKey()——获取公钥函数。
- getK()——获取加密用的参数。
- getPlainText()——获取明文用的函数。
- encryption()——加密函数。
- run()——运行测试函数。

在进行加密之前，首先需要获得公钥(p，g，y)以及加密用的参数 k，然后获取明文，再利用加密函数进行加密。

获取公钥

获取公钥通过函数 getPubKey()来实现，getPubKey()函数的具体代码见程序清单 16-8。

程序清单 16-8

```
void ElgamalB::getPubKey()
{
    ifstream keyIn("pubKey.txt");
    keyIn>>p>>g>>y;
}
```

公钥通过文件“pubKey.txt”获得，具体实现是通过文件输入流完成。

加密参数设置

加密参数设置通过函数 getK()来完成，getK()函数的具体代码见程序清单 16-9。

程序清单 16-9

```
void ElgamalB::getK()
{
    cout<<"Input your private key:";
    while(cin>>k)
    {
        if(k>p-1 || k<=1)
        {
            cout<<"Reinput your private key(1):";
            continue;
```

```
        }
        if(gcd(p-1,k)!=1)
        {
            cout<<"Reinput your private key(2):";
            continue;
        }
        break;
    }
}
```

加密参数根据加密参数的具体要求进行设置，若不满足加密参数的要求，提示用户重新输入相关参数，在获得加密参数中需使用欧几里得函数 gcd()进行判断，gcd()函数的详细代码见程序清单 16-10。

程序清单 16-10

```
int ElgamalB::gcd(int n,int k)
{
    if(n==0)
    {
        return k;
    }
    if(k==0)
    {
        return n;
    }
    return gcd(k, n%k);
}
```

获取明文

明文通过文件获得，具体实现通过函数 getPlainText()完成，getPlainText()函数的具体代码见程序清单 16-11。

程序清单 16-11

```
void ElgamalB::getPlainText()
{
    ifstream plainTextIn("plainText.txt");
    while(plainTextIn>>m)
    {
        if(m>p-1)
        {
            cout<<"File is too long!";
            continue;
        }
        break;
    }
}
```

明文从文件“plainText.txt”获得，在 Elgamal 加密算法中，要求明文的大小要小于 p，因此需对明文的大小进行判断，若不符合要求，需重新进行处理。

加密

加密通过加密函数 encryption()实现，encryption()函数的具体代码见程序清单 16-12。

程序清单 16-12

```
void ElgamalB::encryption()
{
    ofstream cipherOut("cipherText.txt");
    a=expMod(g,k,p);
    word32 temp;
    temp=expMod(y,k,p);
    b=(word64(temp)*m)%p;
    cipherOut<<a<<" "<<b;
}
```

加密过程根据 Elgamal 算法分两步完成，在完成加密后，将加密后的数据 a 和 b 输出到文件“cipherText.txt”，供 A 方解密使用。

16.2.3 算法测试

Elgamal 加密算法测试可以通过 ElgamalA 类与 ElgamalB 类联合执行来完成，在本示例中，ElgamalA 类与 ElgamalB 类分别在两个不同的项目中实现，各自实现各自的功能，ElgamalA 类通过 run()函数来执行密钥生成和解密，run()函数的具体代码见程序清单 16-13。

程序清单 16-13

```
void ElgamalA::run()
{
    int select;
    instruction();
    while(cin>>select)
    {
        switch(select)
        {
            case 0:
                exit(0);
            case 1:
                genPrime();
                break;
            case 2:
                getPrime();
                break;
            case 3:
                genBase();
```

```
                    break;
                case 4:
                    setPrivateKey();
                    break;
                case 5:
                    setPubKey();
                    break;
                case 6:
                    decryption();
                    break;
                default:
                    break;
            }
            instruction();
        }
}
```

在 run()函数中通过变量 select 来确定执行什么操作，若第一次运行，则需要先生成素数 p，然后计算本原根 g，再获取私钥 x，最后计算公钥 g，在获得密文后可以进行解密操作。在 run()函数中使用了函数 instruction()，instruction()函数仅是提示函数，说明不同的选项执行的是什么操作。

在 ElgamalA 类完成密钥生成之后，ElgamalB 就可以执行相应的测试，ElgamalB 也是运行 run()函数进行测试，ElgamalB 的 run()函数的具体代码见程序清单 16-14。

程序清单 16-14

```
void ElgamalB::run()
{
    getPubKey();
    cout<<"p="<<p<<endl;
    cout<<"g="<<g<<endl;
    cout<<"y="<<y<<endl;
    getK();
    cout<<"k="<<k<<endl;
    getPlainText();
    cout<<"m="<<m<<endl;
    encryption();
    cout<<"a="<<a<<endl;
    cout<<"b="<<b<<endl;
}
```

ElgamalB 的 run()函数执行获得加密用密钥、明文等，然后使用 encryption()函数对明文进行加密，并将加密后的数据保存到文件。

在得到 ElgamalB 加密后的数据之后，就可以使用 ElgamalA 的解密函数进行解密，从而完成整个测试工作。

A 方测试结果为

```
Input your selection:
0. Exit.
1. Generate a prime(p).
2. Get a prime(p) from file.
3. Create generator.
4. Create private key.
5. Create public key.
6. Decryption.
Your selection:6
cipherTextA= 52970692
cipherTextB= 354765644
decipherText= 1234562
Now the parameters are:
p=916673803
g=2
x=66666
y=376022059
```

B 方测试结果为

```
p= 916673803
g= 2
y= 376022059
Input your private key:376
Reinput your private key(2):377
k= 377
m= 1234562
a= 52970692
b= 354765644
```

其中,B 方的明文 m 与 A 方脱密后的内容 decipherText 相同。

16.3 习题与实践题

16.3.1 习题

1. 简要说明 Elgamal 加密算法的基本原理。

2. 假设 A,B 双方进行通信,并使用 Elgamal 加密算法,通信过程由 A 方确定密钥,A 方选择的素数 $p=23$,选择的本原根 $g=11$,选择的私钥 $x=7$,B 方加密时选择的 $k=13$,B 方待加密的明文为 15,试给出 A,B 双方进行通信的详细过程。

16.3.2 实践题

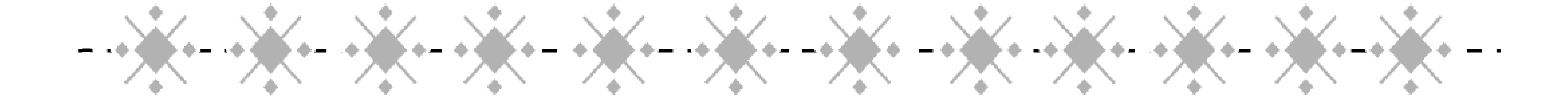
参考 16.2 节的相关内容,编程完成 Elgamal 加密算法,要求:使用一个程序来完成 Elgamal 加密算法,并在程序中能演示 Elgamal 加密算法。

第 部分

散列函数

散列函数又称为哈希函数(Hash Function),它的主要作用是把任意长度的输入通过散列算法变换成固定长度的输出,这个输出被称为散列值或哈希值。散列值的长度要远远小于输入的长度,散列函数就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

在信息安全领域中,比较常用的散列算法包括 MD4、MD5 和 SHA-1 等,MD5 和 SHA-1 算法都是以 MD4 算法为基础设计的,以 MD4 为基础的一系列散列算法有时也称为 MD4 系列算法,在这一部分中,主要介绍 MD4 系列算法以及该系列算法的实现方法。



第17章

MD4算法与MD5算法

MD4算法是由MIT的Ronald L. Rivest于1990年设计的散列算法，MD是Message Digest的缩写，即消息摘要。MD4算法适用于在32位的处理器上使用软件高速实现，主要在32位的操作系统上实现，也是后续许多散列函数的基础，MD5、SHA-1算法都是以MD4算法为基础发展而来，对MD4算法的理解能有效帮助理解MD5和SHA-1等算法。由于MD4算法在安全性上存在一定的缺陷，目前MD4算法已经较少使用。MD5算法是在MD4算法的基础上进行了相应的改进，在文件的完整性检验等方面有着广泛的应用。MD5算法与MD4算法在很大程度上相似，因此，将MD5算法与MD4算法在一章内介绍。

17.1 散列算法基础

17.1.1 散列算法的基本概念

散列函数可以通过以下形式的函数定义：

$$h = H(M) \tag{17-1}$$

h代表散列值或消息摘要(Message Digest)，M代表需计算散列值的消息、文件或其他输入。散列值通常需要提供检查错误的能力，因此，散列值一般是消息M所有位的函数，消息中的任何一位发生变化都将导致散列值发生变化。

在一般情况下，无论输入的消息长度是多长，通过散列算法计算得到的消息摘要或散列值的长度都是固定的，例如，MD5算法输出的消息摘要长度为128位，而SHA-1算法的消息摘要的输出长度为160位。

散列函数的基本工作原理如图17-1所示。

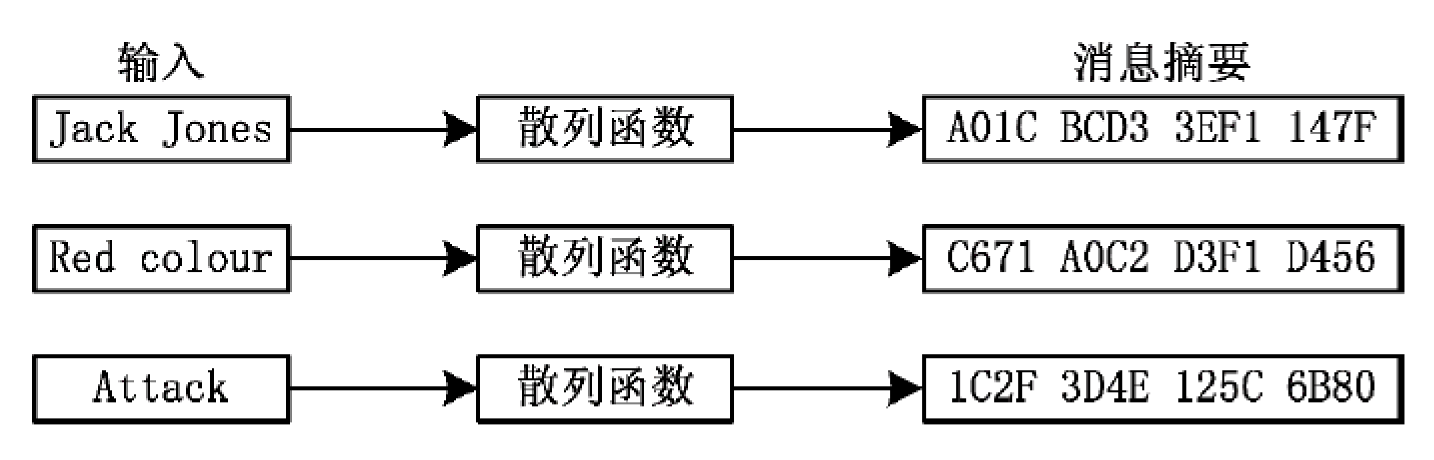


图17-1　散列函数的基本工作原理

散列函数在信息安全领域中主要应用在完整性检测和数字签名中，散列函数在不同的应用中又有不同的名称，当它应用于将任意长度的输入产生固定的输出时被称为压缩(compression)函数，当用于鉴别数据是否被篡改时又被称为数据鉴别码(Data Authetication Code，DAC)或篡改鉴别码(Manipulation Detection Code，MDC)。对于某一种具体的散列算法，不同长度的输入所产生的输出长度是相同的。

一个可用的散列函数需具备以下基本性质：

(1) 对于给定的消息需易于计算其散列值，即已知 M，很容易通过软件或硬件计算得到 $h=H(M)$。

(2) 计算 $h=H(M)$ 得到的散列值的长度是相同的。

(3) 如果消息被改变，那么，其散列值不发生变化是不可能的。即两个不同的消息不可能计算得到相同的散列值。

(4) 如果使用同一个散列算法的两个散列值是不相同的，那么，这两个散列值的原始消息也是不相同的。

一个可用的散列函数除了满足上述的基本性质以外，在计算上还需要满足：

(1) 对于任何给定的散列值 h，找到满足 $H(M)=h$ 的 M 在计算上是不可行的，这个性质也称为单向性，满足该性质的算法也被称为单向散列算法。这条性质使得给定散列值无法计算得到对应的消息。

(2) 对于任何给定的消息 M_1，找到满足 $M_2 \neq M_1$ 且有 $H(M_2)=H(M_1)$ 的 M_2 在计算上是不可行的，这个性质也被称为弱无碰撞性。这条性质使得不能找到散列值相同的另一条消息，可以有效地防止伪造。

(3) 找到任意数据对 (M_2, M_1)，并满足 $H(M_2)=H(M_1)$ 在计算上是不可行的，这个性质也被称为强无碰撞性。这条性质使得生日攻击的方法无法奏效。

散列函数的使用方法比较简单，散列算法的一般结构如图 17-2 所示。

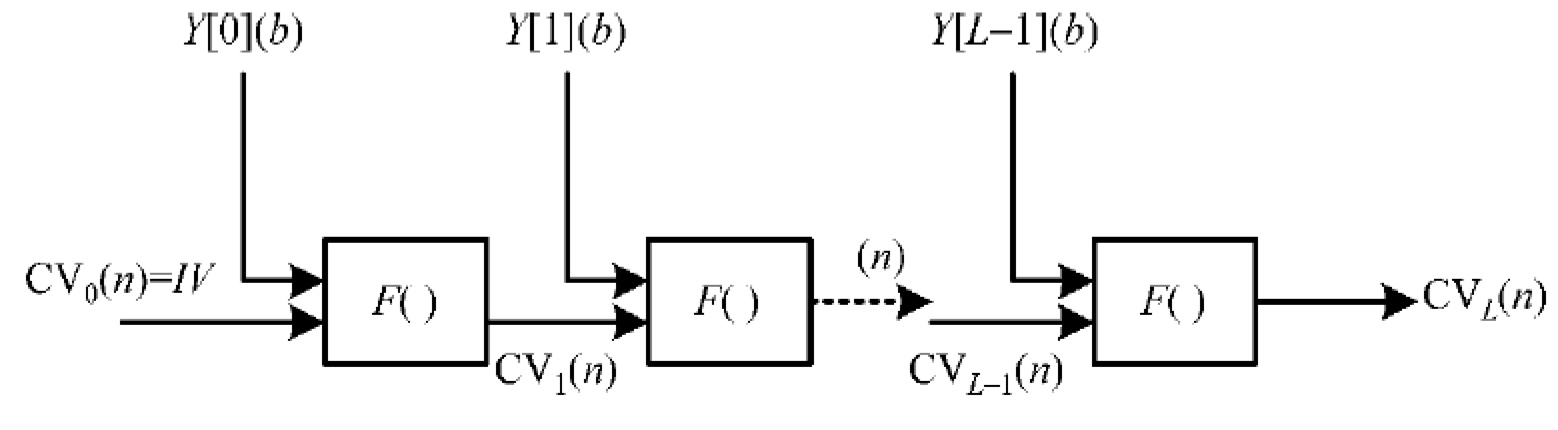


图 17-2 散列算法的一般结构

散列算法通常包括 n 位初始值，即 CV_0，相应的散列函数 $F()$，同时有固定长度(b)的分组输入 $Y[i]$，最后一轮的计算结果为散列值。散列算法也可以表示为

$$\begin{aligned} \mathrm{CV}_0 &= \mathrm{IV} \\ \mathrm{CV}_i &= F(\mathrm{CV}_{i-1}, Y_{i-1}), \quad 1 \leqslant i \leqslant L \\ H(M) &= \mathrm{CV}_L \end{aligned} \tag{17-2}$$

输入的分组长度根据不同的散列算法进行填充，再计算相应的散列值。在每一轮的计算过程中都会用到前面一轮计算得到的散列值，最后一轮计算得到的散列值为输入消息的散列值。

17.1.2 散列算法的使用方法

通过散列算法计算获得消息摘要(散列值)可用于消息鉴别,例如,对文件进行完整性鉴别等,在使用过程中可以根据用途选择不同的使用方法,典型的使用方法共有 6 种模式。

(1)发送方根据消息计算消息摘要,并将消息摘要附加在消息后面,然后对附加消息摘要的消息进行加密,加密完成后再进行发送,其基本使用模式如图 17-3 所示。

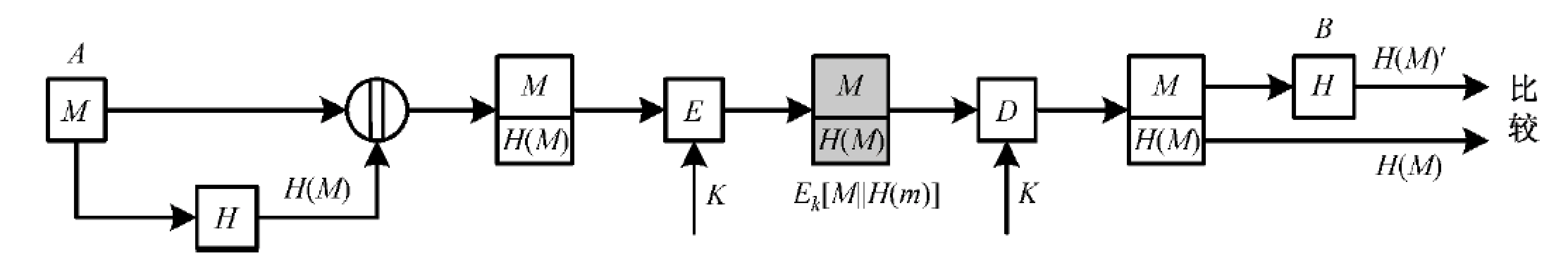


图 17-3 消息摘要处理模式(1)

发送方传送给接收方的数据为

$$A \rightarrow B: E_k[M \mid\mid H(M)]$$

接收方在收到数据之后,对数据进行解密,解密之后将消息与消息摘要分离,然后对明文计算消息摘要,将计算得到的详细摘要与接收到的消息摘要进行比较,若相同则表示消息未被篡改,若不同则表示消息在传递过程中被篡改。采用这种模式进行消息传递时使用的加密算法为对称加密算法,这种模式既包含了消息鉴别,也包含了对消息的加密。

(2) 发送方根据消息计算消息摘要,并对消息摘要进行加密,然后将消息摘要附加在消息的后面,发送给接收方,其基本使用模式如图 17-4 所示。

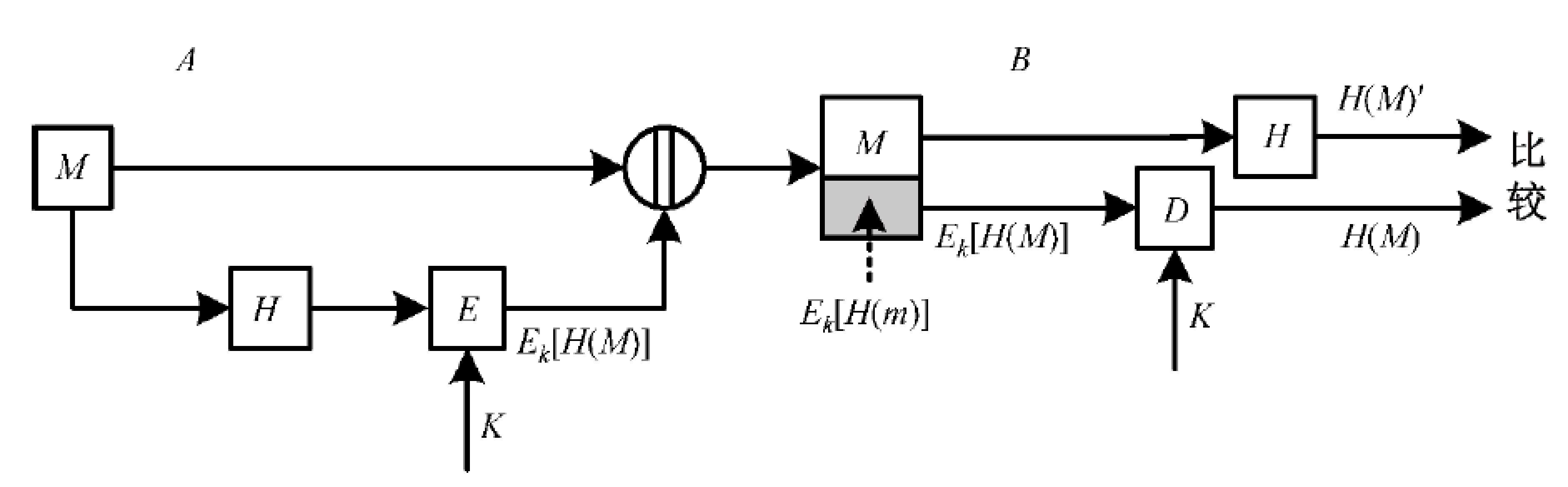


图 17-4 消息摘要处理模式(2)

发送方传送给接收方的数据为

$$A \rightarrow B: M \mid\mid E_k[H(M)]$$

接收方在收到数据之后,将加密的消息摘要和消息分离,对加密的消息摘要进行解密,获得消息摘要,并计算消息的消息摘要,然后与解密得到的消息摘要进行比较,若两者相同则表示消息未被篡改,若不相同,则表示消息被篡改。该模式使用的加密方法是对称加密算法,其主要作用是提供消息的真实性保护。

(3) 第三种模式与第二种模式类似,发送方首先根据消息计算消息摘要,然后使用公钥算法中的私钥对消息摘要进行签名,再将签名后的消息摘要附加在消息的后面,发送给接收方,其基本使用模式如图 17-5 所示。

发送方传送给接收方的数据为

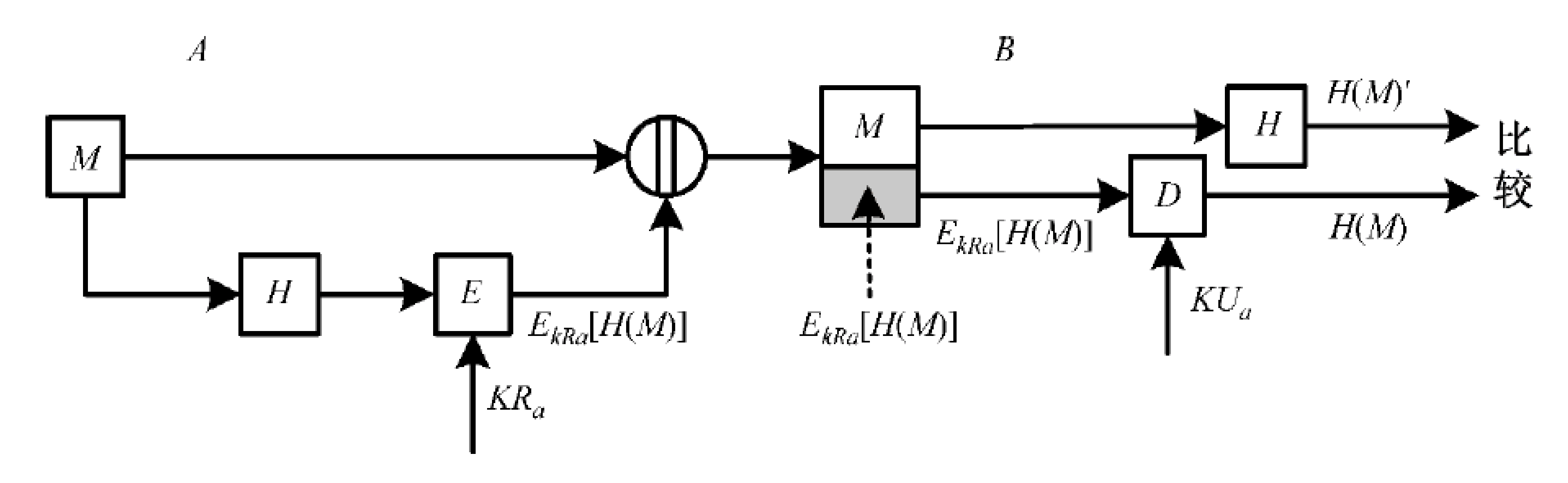


图 17-5 消息摘要处理模式(3)

$$A \to B: M \,||\, E_{kRa}[H(M)]$$

接收方在收到数据之后,将消息与经过签名的消息摘要分离,使用公钥验证签名,并获得消息摘要,同时,重新计算消息的消息摘要,并与原消息摘要进行比较,判断消息是否被篡改。使用这种方法能够达到两个目的:其一是可以判断消息是否被篡改,其二是可以判断消息是否是来自发送方。

(4) 第四种模式既采用了对称加密算法,也采用了公钥加密算法。发送方先根据消息计算消息摘要,然后再使用公钥加密算法中的私钥对消息摘要进行加密,将加密后的消息摘要连接到消息的最后,再使用对称加密算法的密钥进行加密,最后将加密完的数据发送给接收方,其基本使用模式如图 17-6 所示。

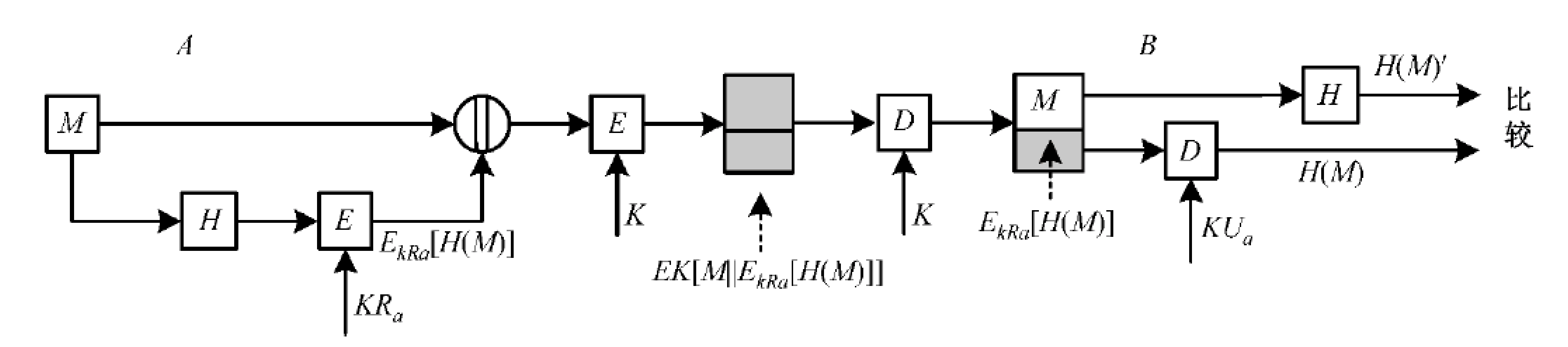


图 17-6 消息摘要处理模式(4)

发送方传送给接收方的数据为

$$A \to B: E_k[M \,||\, E_{kRa}[H(M)]]$$

接收方在接收到发送方的数据后,先使用对称加密算法的密钥对数据进行解密,然后再分离出消息和使用公钥加密算法加密的消息摘要,再使用公钥加密消息摘要,同时根据消息计算消息摘要,最后判断两者是否相同。使用这种模式既提供了对消息的保密,又提供了数字签名,是一种比较常用的模式。

(5) 这种模式在计算散列值之前先将双方共享秘密值连接到消息之后,将计算得到的消息摘要连接到消息之后,并将消息摘要连接到消息最后再将完整数据发送给接收方,其基本使用模式如图 17-7 所示。

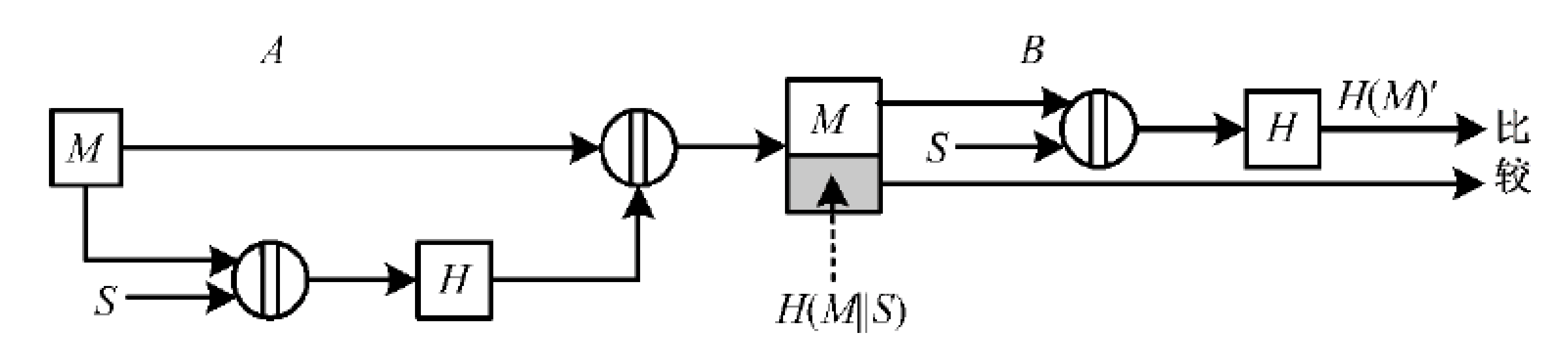


图 17-7 消息摘要处理模式(5)

发送方传送给接收方的数据为

$$A \rightarrow B: M \,||\, H(M \,||\, S)$$

接收方在收到相应数据之后，将消息摘要与消息分离，同时将双方共享秘密值连接到消息之后，再计算消息摘要，最后将两者比较，判断传递的消息是否被篡改。使用这种模式由于在消息的后面连接了双方共享的秘密值，因此，攻击方无法篡改截获的消息，也不能伪造消息。

(6) 第六种处理模式是在第五种处理模式基础上改进而来，同样使用双方共享秘密值连接到消息之后，再计算消息摘要，将计算得到的消息摘要连接到消息最后，再进行加密，使用的加密算法为对称加密算法，再将加密后的数据传送给接收方，其基本使用模式如图 17-8 所示。

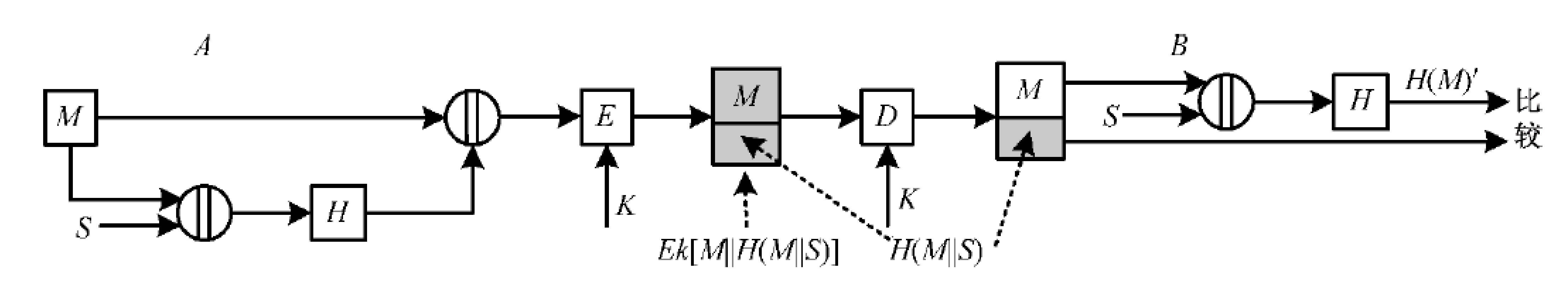


图 17-8 消息摘要处理模式(6)

发送方传送给接收方的数据为

$$A \rightarrow B: E_k[M \,||\, H(M \,||\, S)]$$

接收方在收到消息之后，对收到的数据进行加密，分离消息和消息摘要，然后将双方的秘密值连接到消息之后，再重新计算消息摘要，并与原先得到的消息摘要进行比较，判断得到的消息是否被篡改。

17.2 MD4 算法原理

MD4 算法也称为 MD4 消息摘要算法，MD4 算法的输入是任意长度的消息，输出是 128 位固定长度的消息摘要。在 MD4 算法中，两个不同的消息产生相同的消息摘要在计算上是不可能的，同时，根据消息摘要获得消息在计算上也是不可能的。MD4 算法的设计主要用于数字签名，使用的基本方式是将消息通过 MD4 算法计算获得消息摘要，然后再使用公钥算法中的私钥对消息摘要进行加密，并将消息与消息摘要以一定方式结合发送给接收方。

假设所需处理的消息为 b 位，在 MD4 算法中需满足 $b \geqslant 0$，且 b 是整型数据，其中 b 可以为 0，并且 b 可以不是 8 的倍数（注：ASCII 编码中每个字符长度为 8 位），b 的长度可以是满足条件的任意位数。用 m_i 来表示消息的每一位，那么，b 位的消息可以用如下形式表示：

$$m_0\ m_1\ \cdots\ m_\{b-1\}$$

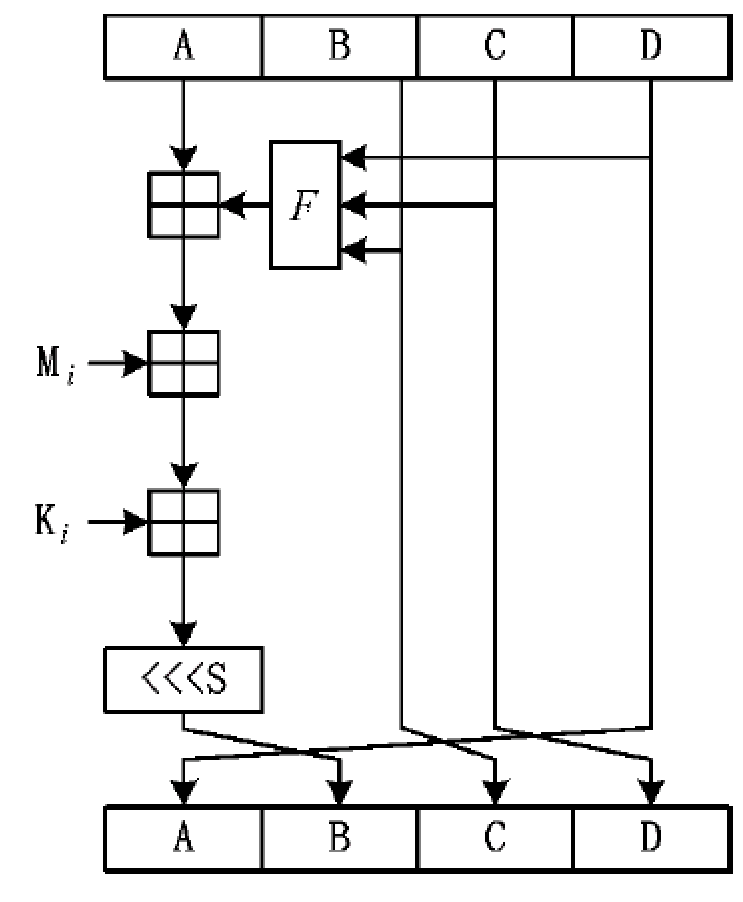


图 17-9 MD4 算法的基本结构

MD4 算法的计算结构如图 17-9 所示。图中，F 表示

非线性运算，每轮进行 16 次相同运算，运算时常量不同，在每轮运算中使用不同的辅助函数进行运算，共进行 3 轮运算，M_i 表示 32 位的分组消息，K_i 表示 32 位的常量，并在每轮运算中各不相同，“ $<<<$ ”表示循环左移。

MD4 算法的整体计算过程可以用图 17-10 来说明。

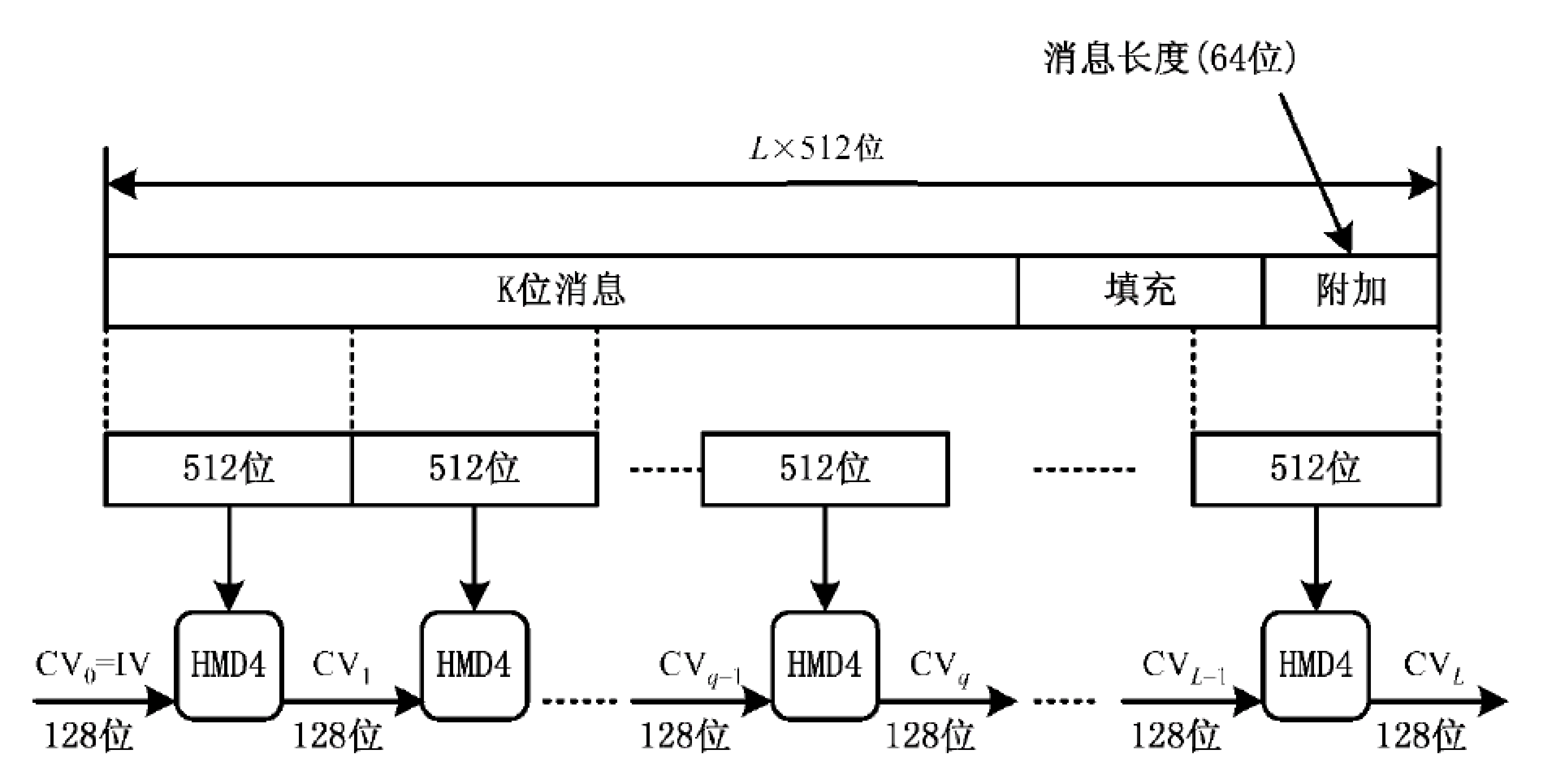


图 17-10 MD4 算法计算过程示意图

MD4 消息摘要的具体计算方法可以通过以下 5 步来完成。

(1) 消息填充

对需要进行消息摘要计算的消息进行填充，使消息的长度符合 length≡448mod512，即填充后的消息的长度是 512 位的整数倍减 64 位。如果初始消息的长度已经符合 length≡448mod512，也需要进行填充。例如，消息的长度正好为 448 位，此时也需要进行填充，填充的长度为 512 位，此时填充完的长度为 960 位。

填充过程的具体方法为：首先在消息最后填充“1”，然后填充“0”，直到经过填充后的消息满足 length≡448mod512 时结束填充。总之，填充的长度最少为 1 位，最长为 512 位。

(2) 填充长度

用 64 位的数据来填充消息 b(消息 b 是指经过第 1 步填充之前的消息)的长度，如果填充后的消息 b 长度大于 2^{64}，那么取其低 2^{64} 位数据。即所包含的消息是对填充前消息的长度关于 2^{64} 进行取模的结果。这 64 位的数据是填充在上一步数据的最后。

通过以上两步得到的消息的长度正好是 512 位的整数倍。如果用 32 位的字(word)来表示，那么，可以等价地表示为 M[01…$N-1$]，其中 N 正好是 16 的整数倍。

示例 17-1 设有二进制格式的输入数据如下：

01100001 01100010 01100011 01100100 01100101

试按照 MD4 算法对数据进行填充。

解 首先在数据最后填充“1”得到：

01100001 01100010 01100011 01100100 01100101 1

然后对数据填充“0”，并达到 448 位结束，用 16 进制表示如下：

61626364　65800000　00000000　00000000
00000000　00000000　00000000　00000000
00000000　00000000　00000000　00000000
00000000　00000000

在最后 64 位填充数据长度，由于数据长度为 40，用 16 进制表示则为 28，因此填充的结果为

61626364　65800000　00000000　00000000
00000000　00000000　00000000　00000000
00000000　00000000　00000000　00000000
00000000　00000000　00000000　00000028

(3) 初始化 MD 缓冲区

在计算 MD4 消息摘要时需要使用 4 字(word)的缓冲(A,B,C,D)，A,B,C,D 可以理解为 4 个 32 位的寄存器，用 16 进制从低位开始的 4 字初始化如下：

```
word A: 01 23 45 67
word B: 89 AB CD EF
word C: FE DC BA 98
word D: 76 54 32 10
```

如果使用整数形式表示，则 4 字初始化的 16 进制结果如下：

```
A= 0x67452301
B= 0xEFCDAB89
C= 0x98BADCFE
D= 0x10325476
```

(4) 以 16 个字(word)的方式处理消息

在以 16 个字的方式处理消息时需要使用 3 个辅助函数，这 3 个辅助函数的输入都是 3 个 32 位的字，输出都是 1 个 32 位的字，这 3 个辅助函数分别如下：

$$F(X,Y,Z) = XY \vee \text{not}(X)\ Z$$
$$G(X,Y,Z) = XY \vee XZ \vee YZ$$
$$H(X,Y,Z) = X \text{ xor } Y \text{ xor } Z$$

在 F,G 和 H 的运算中，使用的运算是按位进行的逻辑运算。XY 表示 $X\&Y$，以此类推。在 3 个辅助函数的基础上，执行以下伪码的运算：

```
for i:=0 to N/16-1
    for j:=0 to 15
        M[i*16+j]:=X[j]
    endfor
    AA=A
    BB=B
    CC=C
    DD=D
/* 第 1轮.*/
```

```
/* [ABCD k s]的操作过程为 A=(A+F(B,C,D)+X[k])<<<s.*/
/* 执行以下 16 步操作 */
    [ABCD  0  3]  [DABC  1  7]  [CDAB  2  11]  [BCDA  3  19]
    [ABCD  4  3]  [DABC  5  7]  [CDAB  6  11]  [BCDA  7  19]
    [ABCD  8  3]  [DABC  9  7]  [CDAB  10 11]  [BCDA  11 19]
    [ABCD  12 3]  [DABC  13 7]  [CDAB  14 11]  [BCDA  15 19]
/* 第 2轮.*/
/* [ABCD k s]的操作过程为 A=(A+G(B,C,D)+X[k]+0x5A827999)<<<s.*/
/* 执行以下 16 步操作 */
    [ABCD  0  3]  [DABC  4  5]  [CDAB  8  9]   [BCDA  12 13]
    [ABCD  1  3]  [DABC  5  5]  [CDAB  9  9]   [BCDA  13 13]
    [ABCD  2  3]  [DABC  6  5]  [CDAB  10 9]   [BCDA  14 13]
    [ABCD  3  3]  [DABC  7  5]  [CDAB  11 9]   [BCDA  15 13]
/* 第 3轮.*/
/* [ABCD k s]的操作过程为 A=(A+H(B,C,D)+X[k]+0x6ED9EBA1)<<<s.*/
/* 执行以下 16 步操作 */
    [ABCD  0  3]  [DABC  8  9]  [CDAB  4  11]  [BCDA  12 15]
    [ABCD  2  3]  [DABC  10 9]  [CDAB  6  11]  [BCDA  14 15]
    [ABCD  1  3]  [DABC  9  9]  [CDAB  5  11]  [BCDA  13 15]
    [ABCD  3  3]  [DABC  11 9]  [CDAB  7  11]  [BCDA  15 15]
    A=A+AA
    B=B+BB
    C=C+CC
    D=D+DD
endfor
```

在执行运算过程中使用了两个 32 位的常量 0x5A827999 和 0x6ED9EBA1。在三轮操作过程中分别使用了 F()函数、G()函数和 H()函数，且每一轮的运算方法略有不同，第二轮和第三轮运算过程中各自增加了一个常量。

(5) 输出

消息摘要的输出是按照 A,B,C,D 的顺序输出，输出以低位 A 作为开始，以高位 D 作为结束。

17.3 MD4 算法实现

MD4 算法的实现过程主要由数据初始化、处理待计算数据的填充、计算哈希值以及输出显示等几部分来完成。

17.3.1 MD4 算法实现的基本结构

本示例主要为解释 MD4 计算哈希值的基本过程，数据的输入直接采用字符串的形式，然后通过读取字符串来进行哈希值的计算，在计算过程中由于大量使用 unsigned int 型数据，因此，将其重定义为 word32，具体定义如下：

```
typedef unsigned int word32;
```

在计算过程中需要存储相应的哈希值、判断待计算数据的长度和计算时用到的临时数据均在结构体 Context 中声明，结构体 Context 的具体声明如下：

```
struct Context
{
    word32 state[4];
    word32 count[2];
    unsigned char buffer[64];
};
```

在结构体 Context 中，变量 state[4]用来存放哈希值，变量 count[2]用来存放待计算哈希值的数据的长度，变量 buffer[64]用来存放计算计算哈希值时用到的临时数据。

MD4 算法的完整内容通过 MD4 类来实现，MD4 类的基本结构如图 17-11 所示。

MD4	
- Padding[64] : unsigned char	
+ MD4 ()	
+ init (Context * context)	: void
+ F (word32 x, word32 y, word32 z)	: word32
+ G (word32 x, word32 y, word32 z)	: word32
+ H (word32 x, word32 y, word32 z)	: word32
+ FF (word32& a, word32 b, word32 c, word32 d, word32 x, word32 s)	: void
+ GG (word32& a, word32 b, word32 c, word32 d, word32 x, word32 s)	: void
+ HH (word32& a, word32 b, word32 c, word32 d, word32 x, word32 s)	: void
+ rotateLeft (word32 x, word32 n)	: word32
+ decode (word32* output, unsigned char* input, word32 len)	: void
+ encode (unsigned char* output, word32* input, word32 len)	: void
+ transformHash (word32 state, unsigned char* block)	: int
+ update (Context* context, unsigned char* input, word32 inputlen)	: int
+ final (unsigned char* digest, Context* context)	: int
+ hashProcss (char* inputString)	: int
+ setBuffer (unsigned char* output, unsigned char* input, word32 len)	: int
+ setMem (unsigned char* output, int value, word32 len)	: int
+ display (unsigned char* digest)	: int

图 17-11 MD4 类的基本结构

MD4 类的具体声明见程序清单 17-1。

程序清单 17-1

```
class MD4
{
    public:
        MD4();
        void init(Context * context);
        word32 F(word32 x,word32 y,word32 z);
        word32 G(word32 x,word32 y,word32 z);
        word32 H(word32 x,word32 y,word32 z);
        void FF(word32 &a,word32 b,word32 c,word32 d,word32 x,word32 s);
```

```
        void GG(word32 &a,word32 b,word32 c,word32 d,word32 x,word32 s);
        void HH(word32 &a,word32 b,word32 c,word32 d,word32 x,word32 s);
        word32 rotateLeft(word32 x,word32 n);
        void decode(word32 * output,unsigned char * input,word32 len);
        void encode(unsigned char * output,word32 * input,word32 len);
        void transformHash(word32 * state,unsigned char * block);
        void update(Context * context,unsigned char * input,word32 inputlen);
        void final(unsigned char * digest,Context * context);
        void hashProcss(char * inputString);
        void setBuffer(unsigned char * output,unsigned char * input,word32 len);
        void setMem(unsigned char * output,int value,word32 len);
        void display(unsigned char * digest);
    private:
        unsigned char Padding[64];
};
```

MD4 类中的成员变量 Padding[64]用于填充数据，其初始化是在 MD4 类的构造函数中进行，MD4 类中的各成员函数的具体作用如下：

- MD4()——构造函数，用于初始化相关成员变量。
- init()——初始化函数，用于初始化计算哈希值的相关变量。
- F()、G()、H()——计算哈希值中的变换函数。
- FF()、GG()、HH()——计算哈希值中的变换函数。
- rotateLeft()——循环左移函数。
- decode()——编码转换函数。
- encode()——编码转换函数。
- transformHash()——哈希值计算函数。
- update()——计算哈希值用的辅助函数。
- final()——处理哈希值的函数。
- hashProcss()——运行哈希值计算的函数。
- setBuffer()——处理计算哈希值过程中使用的缓存的函数。
- setMem()——哈希值计算过程中变量转换的函数。
- display()——显示最终哈希值计算结果的函数。

17.3.2 数据初始化

MD4 类的数据初始化分别由两个函数来执行，其一是构造函数，另一个是初始化函数，构造函数 MD4()的详细代码见程序清单 17-2。

程序清单 17-2

```
MD4::MD4()
{
    Padding[0]=0x80;
    word32 i;
    for(i=1;i<64;i++)
```

```
06      {
07          Padding[i]=0;
08      }
09  }
```

构造函数主要用于填充数据的初始化,第 1 个填充数据为 0x80,见代码行第 3 行,其余填充数据均为 0,在本示例中填充数据采用的是普通变量来处理,也可以将填充数据用的相关变量设为常量。

另一个初始化数据用的函数 init()的详细代码见程序清单 17-3。

程序清单 17-3

```
void MD4::init(Context * context)
{
    context->state[0]=0x67452301;
    context->state[1]=0xefcdab89;
    context->state[2]=0x98badcfe;
    context->state[3]=0x10325476;
    context->count[0]=0;
    context->count[1]=0;
}
```

init()函数的参数为结构 Context 类型,init()函数主要初始化计算哈希值用的变量 state[]和长度变量 count[],而结构体的另一个变量 buffer[]并没有在 init()函数中初始化,变量 buffer[]在每次使用中需单独重新赋值,因此,不需要在初始化过程中进行复制,若要对变量 buffer[]进行赋值,只需要将变量 buffer[]中的各个元素的值设为 0 即可。

17.3.3 辅助函数的实现

在哈希值的计算过程中使用了一些相关的辅助函数,包括一些基本运算函数,数据转换函数,这些函数是实现哈希值计算过程的一些基本运算和基本数据转换。

在每一轮的哈希值的计算过程中都会使用到变换函数 F()、G()和 H()函数,这三个函数是 MD4 算法中的基本函数,所进行的运算都是位运算,实现的方法都比较简单。

F()函数的详细实现代码见程序清单 17-4。

程序清单 17-4

```
word32 MD4::F(word32 x,word32 y,word32 z)
{
    return (((x) & (y)) | ((~ x) &(z)));
}
```

F()函数的参数和返回类型都是 word32 型,函数根据相应的参数进行位运算,然后将相应的计算结果返回。

G()函数的详细实现代码见程序清单 17-5。

程序清单 17-5

```
word32 MD4::G(word32 x,word32 y,word32 z)
```

```
{
    return (((x) & (y))|((x)& (z))|((y) & (z)));
}
```

G()函数的参数和返回类型与 F()函数相同，也都是 word32 型，同样也是进行相应的位运算，并将相应的计算结果返回。

H()函数的详细实现代码见程序清单 17-6。

程序清单 17-6

```
word32 MD4::H(word32 x,word32 y,word32 z)
{
    return ((x)^(y)^(z));
}
```

H()函数的参数和返回类型也与 F()函数相同，也都是 word32 型，同样也是进行相应的位运算，并将相应的计算结果返回。

在哈希值的计算过程中还用到了另外三个基本运算函数——FF()函数、GG()函数和 HH()函数，这三个函数分别与 F()函数，G 函数和 H()函数相对应，并以这三个函数作为运算的基础。

FF()函数的详细实现代码见程序清单 17-7。

程序清单 17-7

```
void MD4::FF(word32 &a,word32 b,word32 c,word32 d,word32 x,word32 s)
{
    a+=F(b,c,d)+x;
    a=rotateLeft(a,s);
}
```

函数的参数均为 word32 型数据，a，b，c，d 分别为计算哈希值的各变量，每个为 32 位，共 128 位，参数 a 使用的是引用型变量，计算得到的结果自动返回。参数 x 和 s 为每轮运算过程中的变量，参数 s 实际上可以看做是常量。在计算过程中使用了 F()函数，并将计算结果与变量 x 的值相加，然后进行循环移位，完成一轮的操作。

在 FF()函数中用到了循环左移函数 rotateLeft()，该函数实现了循环左移的功能，rotateLeft()函数的详细实现代码见程序清单 17-8。

程序清单 17-8

```
word32 MD4::rotateLeft(word32 x,word32 n)
{
    return (((x)<< (n))|((x)>> (32- (n))));
}
```

rotateLeft()函数的参数和返回值都是 word32 型，函数的第 1 个参数是要进行循环左移的变量，第 2 个参数是所要进行循环左移的量，循环左移后的结果作为返回值。

GG()函数的详细实现代码见程序清单 17-9。

程序清单 17-9

```
void MD4::GG(word32 &a,word32 b,word32 c,word32 d,word32 x,word32 s)
{
    a+=G(b,c,d)+x+0x5A827999;
    a=rotateLeft(a,s);
}
```

GG()函数在实现过程中使用 G()函数，同时还用了常量 0x5A827999。函数参数的含义与 FF()函数相同，变量 a 也是引用型变量，计算结果也是自动返回，在计算过程中也使用了循环左移函数 rotateLeft()。

HH()函数的详细实现代码见程序清单 17-10。

程序清单 17-10

```
void MD4::HH(word32 &a,word32 b,word32 c,word32 d,word32 x,word32 s)
{
    a+=H(b,c,d)+x+0x6ED9EBA1;
    a=rotateLeft(a,s);
}
```

HH()函数的实现方法与 GG()函数类似，在计算过程使用了另一个常量 0x6ED9EBA1，函数参数的含义与 GG()函数相同，同样也使用了循环左移函数 rotateLeft()。

以上六个函数实现了 MD4 算法的基本运算，在 MD4 算法的实现过程中还需要在 word32 型数据与 unsigned char 型数据之间互相转化，转化功能通过函数 decode()和函数 encode()来实现，函数 decode()的详细代码见程序清单 17-11。

程序清单 17-11

```
void MD4::decode(word32 * output,unsigned char * input,word32 len)
{
    word32 i,j;
    for(i=0,j=0;j<len;i++,j+=4)
    {
        output[i]=((word32)input[j])|(((word32)input[j+1])<<8)
                |(((word32)input[j+2])<<16)|(((word32)input[j+3])<<24);
    }
}
```

decode()函数的参数是 word32 指针型作为输出，unsigned char 指针型作为输入，同时还有 word32 型的参数，该参数是输入数据的长度，函数的功能是将 unsigned char 型输入数据转化为 word32 型数据，其实现的原理如图 17-12 所示。

在进行移位之前需要将 unsigned char 型数据强制转化为 word32 型数据，在转化完成之后再进行移位操作，并将移位后的数据进行“或”运算，得到最终的 word32 型数据。

encode()函数的实现过程正好与 decode()函数的实现相反，encode()函数的详细实现代码见程序清单 17-12。

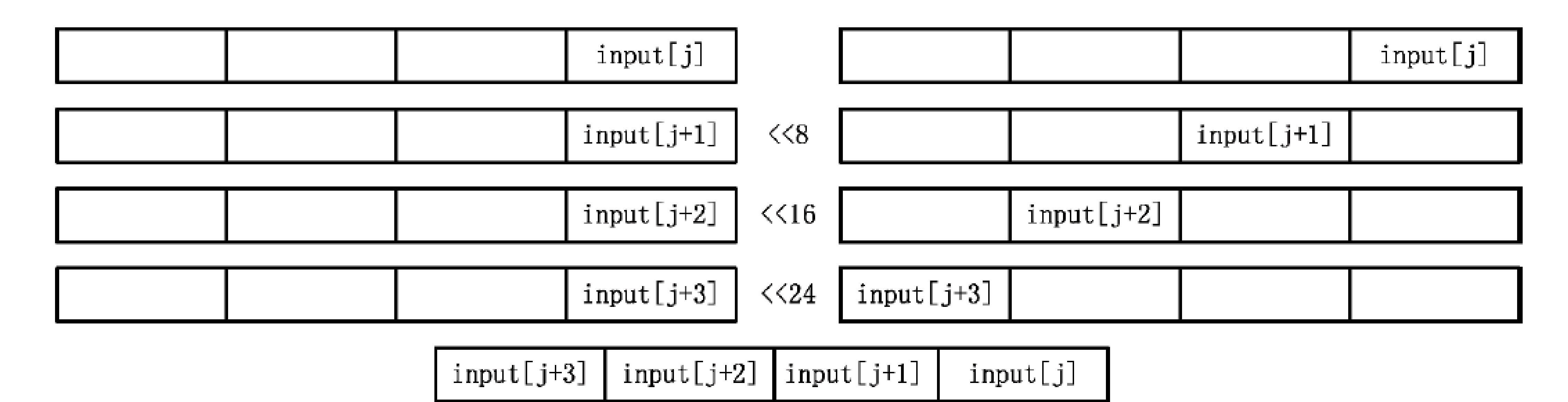


图 17-12　数据转化示意图

程序清单 17-12

```
void MD4::encode(unsigned char * output,word32 * input,word32 len)
{
    word32 i,j;
    for(i=0,j=0;j<len;i++,j+=4)
    {
        output[j]=(unsigned char)(input[i]&0xFF);
        output[j+1]=(unsigned char)(input[i]>>8&0xFF);
        output[j+2]=(unsigned char)(input[i]>>16&0xFF);
        output[j+3]=(unsigned char)(input[i]>>24&0xFF);
    }
}
```

encode()是以 word32 型数据作为输入，unsigned char 型数据作为输出。每次的运算过程将 word32 型输入与“0xFF”进行“&”运算，并将获取的数据强制转化为 unsigned char 型数据，获得第一个数据。然后右移 8 位，再进行同样的操作，获取下一个数据，以此类推，直至获取所有数据。

17.3.4　哈希值计算过程的实现

transformHash()函数是哈希值计算过程的核心函数，transformHash()函数完成哈希值计算的基本过程，函数的详细实现代码见程序清单 17-13。

程序清单 17-13

```
void MD4::transformHash(word32 * state,unsigned char * block)
{
    word32 A,B,C,D;
    word32 x[16];
    A=state[0];
    B=state[1];
    C=state[2];
    D=state[3];
    decode(x,block,64);
    FF(A,B,C,D,x[0],3);        FF(D,A,B,C,x[1],7);
    FF(C,D,A,B,x[2],11);       FF(B,C,D,A,x[3],19);
```

```
        FF(A,B,C,D,x[4],3);         FF(D,A,B,C,x[5],7);
        FF(C,D,A,B,x[6],11);        FF(B,C,D,A,x[7],19);
        FF(A,B,C,D,x[8],3);         FF(D,A,B,C,x[9],7);
        FF(C,D,A,B,x[10],11);       FF(B,C,D,A,x[11],19);
        FF(A,B,C,D,x[12],3);        FF(D,A,B,C,x[13],7);
        FF(C,D,A,B,x[14],11);       FF(B,C,D,A,x[15],19);
        GG(A,B,C,D,x[0],3);         GG(D,A,B,C,x[4],5);
        GG(C,D,A,B,x[8],9);         GG(B,C,D,A,x[12],13);
        GG(A,B,C,D,x[1],3);         GG(D,A,B,C,x[5],5);
        GG(C,D,A,B,x[9],9);         GG(B,C,D,A,x[13],13);
        GG(A,B,C,D,x[2],3);         GG(D,A,B,C,x[6],5);
        GG(C,D,A,B,x[10],9);        GG(B,C,D,A,x[14],13);
        GG(A,B,C,D,x[3],3);         GG(D,A,B,C,x[7],5);
        GG(C,D,A,B,x[11],9);        GG(B,C,D,A,x[15],13);
        HH(A,B,C,D,x[0],3);         HH(D,A,B,C,x[8],9);
        HH(C,D,A,B,x[4],11);        HH(B,C,D,A,x[12],15);
        HH(A,B,C,D,x[2],3);         HH(D,A,B,C,x[10],9);
        HH(C,D,A,B,x[6],11);        HH(B,C,D,A,x[14],15);
        HH(A,B,C,D,x[1],3);         HH(D,A,B,C,x[9],9);
        HH(C,D,A,B,x[5],11);        HH(B,C,D,A,x[13],15);
        HH(A,B,C,D,x[3],3);         HH(D,A,B,C,x[11],9);
        HH(C,D,A,B,x[7],11);        HH(B,C,D,A,x[15],15);
        state[0]+=A;
        state[1]+=B;
        state[2]+=C;
        state[3]+=D;
    }
```

transformHash()函数的核心是分别进行 16 轮的 FF(),GG()和 HH()运算,函数的参数一个是哈希值,另一个是需计算哈希值的分组,两个参数均为指针型,在计算之前需要将输入的 unsigned char 型分组转化为 word32 型数组,在程序中使用了 decode()函数来实现该功能。

根据输入的数据长度进行填充和哈希值的计算过程分别由 update()函数和 final()函数完成,update()函数与 final()函数结合计算哈希值的过程可以描述为:

(1) 如果输入数据的长度超过 64 字节(512 位),则每 64 字节计算哈希值,至数据长度小于 64 字节结束。

(2) 对上一步剩余部分数据进行填充,如果填充后的长度大于 64 字节,则对 64 字节计算哈希值,否则进入下一步。

(3) 对上一步得到的数据进行输入数据长度填充,并对填充后的数据计算哈希值,得到数据最终哈希值的计算结果。

update()函数的详细代码见程序清单 17-14。

程序清单 17-14

```
void MD4::update(Context * context,unsigned char * input,word32 inputlen)
```

```
02 {
03     word32 i,index,partLength;
04     index=(word32)((context->count[0]>>3)&0x3F);
05     if((context->count[0]+=((word32)inputlen<<3))<((word32)inputlen<<3))
06     {
07         context->count[1]++;
08     }
09     context->count[1]+=((word32)inputlen>>29);
10     partLength=64-index;
11     if(inputlen>=partLength)
12     {
13         setBuffer((unsigned char *)&context->buffer[index],
14                 (unsigned char *)input,partLength);
15         transformHash(context->state,context->buffer);
16         for(i=partLength;i+63<inputlen;i+=64)
17         {
18             transformHash(context->state,&input[i]);
19         }
20         index=0;
21     }
22     else
23     {
24         i=0;
25     }
26     setBuffer((unsigned char *)&context->buffer[index],
27             (unsigned char *)&input[i],inputlen-i);
28 }
```

程序中的第 4 行代码是用于计算输入数据的字节数模 64，由于 Context 结构体中的 count 变量是用于计算输入数据的总位数(bit)，将该变量转换成字节总数需要将总位数除 8，该运算相当于将对应变量右移 3 位(>>3)，而 &0x3F 运算则是进行模 64 运算。经过这步运算得到输入数据的字节数模 64。第 5 行代码到第 9 行代码是将待计算哈希值的字节型数据(byte)的长度转换为以位(bit)为单位的 Context 结构体的 count 数组中。

final()函数的详细代码见程序清单 17-15。

程序清单 17-15

```
void MD4::final(unsigned char * digest,Context * context)
{
    unsigned char bits[8];
    word32 index,padLength;
    encode(bits,context->count,8);
    index=(word32)((context->count[0]>>3)&0x3F);
    padLength=(index<56)?(56-index):(120-index);
    update(context,Padding,padLength);
    update(context,bits,8);
```

```
10        encode(digest,context->state,16);
11        setMem((unsigned char * )context,0,sizeof(* context));
12    }
```

update()函数主要处理输入长度大于 64(512 位)时进行哈希值预计算，而 final()函数则根据输入的长度确定需要填充数据的长度，在进行填充后再调用 update()函数，计算相应哈希值，然后将计算结果转化输出。第 7 行代码用于确定需要填充的数据长度，由于在一个分组中有 8 字节(64 位)用于附加输入数据的总长度，因此，在计算需要填充的数据长度时是以 64－8＝56 作为判断的标准。

在计算哈希值的过程中使用了 setBuffer()函数，该函数是将所需计算的数据转存到 Context 结构体 buffer[64]数组中，setBuffer()函数的详细实现代码见程序清单 17-16。

程序清单 17-16

```
void MD4::setBuffer(unsigned char * output,unsigned char * input,word32 len)
{
    word32 i;
    for(i=0;i<len;i++)
    {
        output[i]=input[i];
    }
}
```

setBuffer()函数实现的功能比较简单，只是将输入的数据根据所需的长度转存到 buffer[64]数组中。

在完成所有计算之后，通过 setMem()函数重置 Context 结构体，setMem()函数的具体代码见程序清单 17-17。

程序清单 17-17

```
void MD4::setMem(unsigned char * output,int value,word32 len)
{
    word32 i;
    for(i=0;i<len;i++)
    {
        ((char* )output)[i]=(char)value;
    }
}
```

该函数的第 3 个参数是结构体的大小，通过 sizeof(* context)获得。

17.3.5 测试与输出

程序的驱动通过函数 hashProcss()进行，通过 display()函数将计算结果显示出来，在具体执行过程中，只需要通过 main()函数调用 hashProcss()函数就可以完成测试。

hashProcss()函数的详细代码见程序清单 17-18。

程序清单 17-18

```
void MD4::hashProcss(char * inputString)
{
    cout<< "hash("<< inputString<< ")= ";
    Context context;
    init(&context);
    unsigned char digest[16];
    word32 len= strlen(inputString);
    update(&context,(unsigned char * )inputString,len);
    final(digest,&context);
    display(digest);
}
```

hashProcss()函数的参数是输入的字符串,函数中的变量 digest[16]用来存储计算所得的哈希值,由于在计算哈希值过程中使用的变量类型为 unsigned char,因此,在计算前需将输入的 char 型字符串转换为 unsigned char 型。

在计算完成之后,通过 display()将计算结果显示出来,display()函数的详细代码见程序清单 17-19。

程序清单 17-19

```
01 void MD4::display(unsigned char * digest)
02 {
03     word32 i;
04     for(i= 0;i< 16;i+ + )
05     {
06         cout<< hex<< (int)digest[i];
07     }
08     cout<< endl;
09 }
```

哈希值通过 16 进制的方式进行显示,见代码行第 6 行。

具体测试过程通过 main()函数执行,main()函数的具体代码见程序清单 17-20。

程序清单 17-20

```
int main()
{
    MD4 md4;
    md4.hashProcss("");
    md4.hashProcss("a");
    md4.hashProcss("abcdefghijklmnopqrstuvwxyz");
    md4.hashProcss("12345678901234567890123456789012345678901234567890123");
    md4.hashProcss("The quick brown fox jumps over the lazy cog");
    md4.hashProcss("The quick brown fox jumps over the lazy dog");
    return 0;
}
```

测试的结果如下：

```
hash()=31d6cfe0d16ae931b73c59d7e0c089c0
hash(a)=bde52cb31de33e46245e5fbdbd6fb24
hash(abcdefghijklmnopqrstuvwxyz)=d79e1c308aa5bbcdeea8ed63df412da9
hash(12345678901234567890123456789012345678901234567890123456789012345678901234567890)=5433cc73c8b542abc409336e6e8081
hash(The quick brown fox jumps over the lazy cog)=b86e130ce728da59e672d56ad113df
hash(The quick brown fox jumps over the lazy dog)=1bee69a46ba811185c194762abaeae90
```

测试数据1,2,3采用了文献[66]中的测试数据,测试结果与文献[66]一致。第5组和第6组测试数据是一组简单对比,仅将倒数第3个字符进行了修改,得到的哈希值完全不同。

17.4 MD5 算法原理

MD5消息摘要算法也是由Ron Rivest提出的,MD5算法实际上是MD4算法的改进版本。MD5算法的输入可以是任意长度的消息,输出是128位的消息摘要。MD5算法与MD4算法相比主要增加了算法的复杂性和不可逆性。

目前,MD5算法比较广泛应用于完整性检验,例如,在服务器上有待下载文件,并有计算得到的哈希值,用户下载完文件后可以使用MD5算法来计算文件的哈希值,并与服务器上提供的哈希值进行比较,若两者相同,则表示文件在传输过程中没有被篡改,否则,下载的文件可能被篡改或不完整。

MD5算法的基本结构与MD4算法的基本结构类似,并在MD4算法的基础上进行了一些改进,MD5算法的基本结构如图17-13所示。

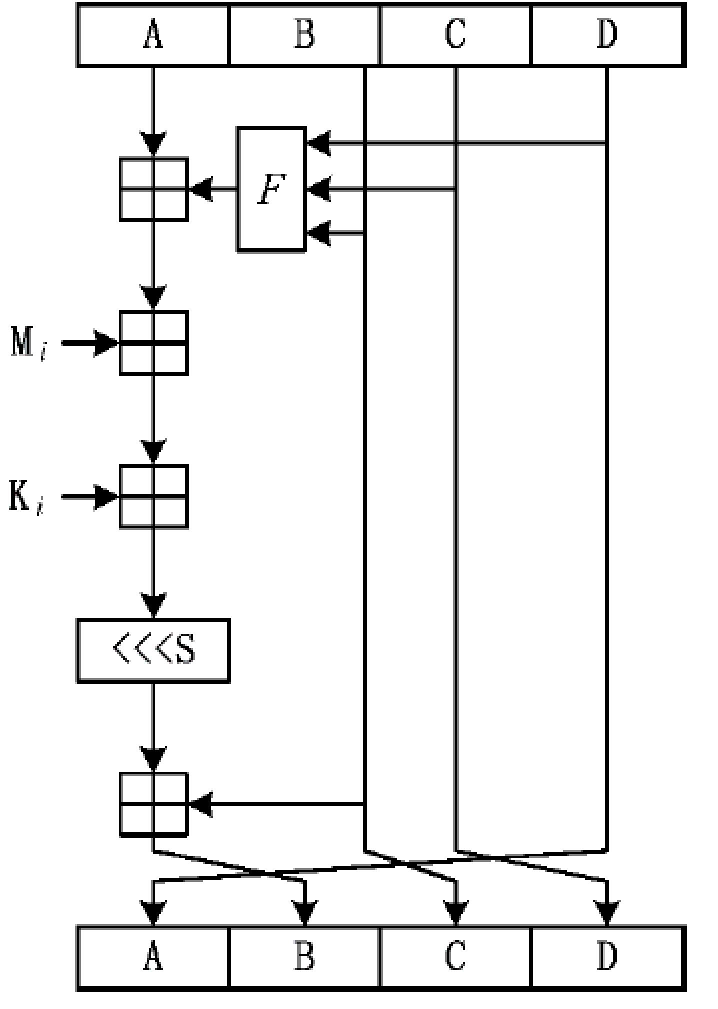


图17-13 MD5算法的基本结构

与MD4算法相比,在一轮运算中,A在输出之前还进行加B模2^{32}运算,同时在每轮运算过程中,运算的基本方法也做了适当的改进。MD5算法计算消息摘要的基本过程如下。

(1) 预处理

MD5算法的预处理过程与MD4算法相同,同样包括消息填充、长度填充和初始化MD缓冲区,具体参考MD4的这部分内容。

(2) 以16个字(word)的方式处理消息

在以16个字的方式处理消息时需要使用4个辅助函数,这4个辅助函数的输入都是3个32位的字,输出都是1个32位的字,这4个辅助函数分别如下:

$F(X,Y,Z)=XY \vee \text{not}(X)\ Z$

$G(X,Y,Z)=XZ \vee Y\,\text{not}(Z)$

$H(X,Y,Z)=X \text{ xor } Y \text{ xor } Z$

$I(X,Y,Z)=Y \text{ xor } (X \vee \text{not}(Z))$

在 F,G,H 和 I 的运算中,使用的运算是按位进行的逻辑运算。XY 表示 $X\&Y$,以此类推。与 MD4 算法相比增加了一个 I 运算,同时,对原来的 G 函数的运算方法进行了适当的修改。在 3 个辅助函数的基础上,执行以下伪码的运算:

```
for i:=0 to N/16-1
    for j:=0 to 15
        M[i*16+j]:=X[j]
    endfor
    AA=A
    BB=B
    CC=C
    DD=D
    /* 第 1 轮. */
    /* 设 [ABCD k s i]的操作为 A=B+((A+F(B,C,D)+X[k]+T[i])<<<s). */
    /* 执行以下 16次操作. */
    [ABCD  0  7  1]  [DABC  1 12  2]  [CDAB  2 17  3]  [BCDA  3 22  4]
    [ABCD  4  7  5]  [DABC  5 12  6]  [CDAB  6 17  7]  [BCDA  7 22  8]
    [ABCD  8  7  9]  [DABC  9 12 10]  [CDAB 10 17 11]  [BCDA 11 22 12]
    [ABCD 12  7 13]  [DABC 13 12 14]  [CDAB 14 17 15]  [BCDA 15 22 16]
    /* 第 2轮. */
    /* 设 [ABCD k s i]的操作为 A=B+((A+G(B,C,D)+X[k]+T[i])<<<s). */
    /* 执行以下 16次操作. */
    [ABCD  1  5 17]  [DABC  6  9 18]  [CDAB 11 14 19]  [BCDA  0 20 20]
    [ABCD  5  5 21]  [DABC 10  9 22]  [CDAB 15 14 23]  [BCDA  4 20 24]
    [ABCD  9  5 25]  [DABC 14  9 26]  [CDAB  3 14 27]  [BCDA  8 20 28]
    [ABCD 13  5 29]  [DABC  2  9 30]  [CDAB  7 14 31]  [BCDA 12 20 32]
    /* 第 3轮. */
    /* 设 [abcd k s t]的操作为 a=b+((a+H(b,c,d)+X[k]+T[i])<<<s). */
    /* 执行以下 16次操作. */
    [ABCD  5  4 33]  [DABC  8 11 34]  [CDAB 11 16 35]  [BCDA 14 23 36]
    [ABCD  1  4 37]  [DABC  4 11 38]  [CDAB  7 16 39]  [BCDA 10 23 40]
    [ABCD 13  4 41]  [DABC  0 11 42]  [CDAB  3 16 43]  [BCDA  6 23 44]
    [ABCD  9  4 45]  [DABC 12 11 46]  [CDAB 15 16 47]  [BCDA  2 23 48]
    /* 第 4轮. */
    /* 设 [ABCD k s t]的操作为 A=B+((A+I(B,C,D)+X[k]+T[i])<<<s). */
    /* 执行以下 16次操作. */
    [ABCD  0  6 49]  [DABC  7 10 50]  [CDAB 14 15 51]  [BCDA  5 21 52]
    [ABCD 12  6 53]  [DABC  3 10 54]  [CDAB 10 15 55]  [BCDA  1 21 56]
    [ABCD  8  6 57]  [DABC 15 10 58]  [CDAB  6 15 59]  [BCDA 13 21 60]
    [ABCD  4  6 61]  [DABC 11 10 62]  [CDAB  2 15 63]  [BCDA  9 21 64]
    A=A+AA
    B=B+BB
    C=C+CC
    D=D+DD
endfor
```

在 MD5 算法的四轮计算过程中分别使用辅助函数 F(),G(),H()和 I(),与 MD 算法相比,除了使用的 X[k]不同外,还增加了另外一个常量 T[i],T[i]在每一次运算过程中各不相同,其具体的值在程序中给出。

(3) 输出

消息摘要的输出是按照 A,B,C,D 的顺序输出,输出以低位 A 作为开始,以高位 D 作为结束。

17.5　MD5 算法实现

MD5 算法的实现过程主要由数据初始化、处理待计算数据的填充、计算哈希值以及输出显示等几部分来完成。

17.5.1　MD5 算法实现的基本结构

为便于与 MD4 算法的实现进行比较,MD5 算法的实现可以采用与 MD4 算法相似的结构,基本数据仍采用 word32 来处理,哈希值计算同样采用结构体进行,实现 MD5 算法的类 MD5 的基本结构如图 17-14 所示。

MD5	
- Padding[64]	: unsigned char
- FS[16]	: static const word32
- GS[16]	: static const word32
- HS[16]	: static const word32
- IS[16]	: static const word32
- FAC[16]	: static const word32
- GAC[16]	: static const word32
- HAC[16]	: static const word32
- IAC[16]	: static const word32
+ MD5 ()	
+ init (Context * context)	: void
+ F (word32 x, word32 y, word32 z)	: word32
+ G (word32 x, word32 y, word32 z)	: word32
+ H (word32 x, word32 y, word32 z)	: word32
+ I (word32 x, word32 y, word32 z)	: word32
+ FF (word32& a, word32 b, word32 c, word32 d, word32 x, const word32 s, const word32 ac)	: void
+ GG (word32& a, word32 b, word32 c, word32 d, word32 x, const word32 s, const word32 ac)	: void
+ HH (word32& a, word32 b, word32 c, word32 d, word32 x, const word32 s, const word32 ac)	: void
+ II (word32& a, word32 b, word32 c, word32 d, word32 x, const word32 s, const word32 ac)	: void
+ rotateLeft (word32 x, word32 n)	: word32
+ decode (word32* output, unsigned char* input, word32 len)	: void
+ encode (unsigned char* output, word32* input, word32 len)	: void
+ transformHash (word32 state, unsigned char* block)	: int
+ update (Context* context, unsigned char* input, word32 inputlen)	: int
+ final (unsigned char* digest, Context* context)	: int
+ hashProcss (char* inputString)	: int
+ setBuffer (unsigned char* output, unsigned char* input, word32 len)	: int
+ setMem (unsigned char* output, int value, word32 len)	: int
+ display (unsigned char* digest)	: int

图 17-14　MD5 类的基本结构

与 MD4 类相比，MD5 类增加了 FS[16]，GS[16]，HS[16]和 IS[16]轮运算中的常量，同时也增加了 FAC[16]，GAC[16]，HAC[16]和 IAC[16]轮运算常量，同时，FF()，GG()和 HH()函数参数增加一个参数，并增加了 II()函数用于第 4 轮运算。

MD5 类的声明见程序清单 17-21。

程序清单 17-21

```
class MD5
{
    public:
        MD5();
        void init(Context * context);
        word32 F(word32 x,word32 y,word32 z);
        word32 G(word32 x,word32 y,word32 z);
        word32 H(word32 x,word32 y,word32 z);
        word32 I(word32 x,word32 y,word32 z);
        void FF(word32 &a,word32 b,word32 c,word32 d,
            word32 x,const word32 s,const word32 ac);
        void GG(word32 &a,word32 b,word32 c,word32 d,
            word32 x,const word32 s,const word32 ac);
        void HH(word32 &a,word32 b,word32 c,word32 d,
            word32 x,const word32 s,const word32 ac);
        void II(word32 &a,word32 b,word32 c,word32 d,
            word32 x,word32 s,word32 ac);
        word32 rotateLeft(word32 x,word32 n);
        void decode(word32 * output,unsigned char * input,word32 len);
        void encode(unsigned char * output,word32 * input,word32 len);
        void transformHash(word32 * state,unsigned char * block);
        void update(Context* context,unsigned char * input,word32 inputlen);
        void final(unsigned char* digest,Context * context);
        void hashProcss(char * inputString);
        void setBuffer(unsigned char * output,unsigned char * input,word32 len);
        void setMem(unsigned char * output,int value,word32 len);
        void display(unsigned char* digest);
    private:
        unsigned char Padding[64];
        static const word32 FS[16];
        static const word32 GS[16];
        static const word32 HS[16];
        static const word32 IS[16];
        static const word32 FAC[16];
        static const word32 GAC[16];
        static const word32 HAC[16];
        static const word32 IAC[16];
};
```

与 MD4 类相比,代码行第 9 行是增加的 I() 函数,代码行第 16 行增加了 II() 函数,代码行第 30 行到第 37 行增加的常量分别是用于四轮计算过程中使用的常量。

17.5.2 数据初始化

MD5 类的数据初始化包括 Context 结构体相关数据的初始化、填充数据的初始化和计算哈希值过程中使用的各常量的初始化,填充数据的初始化通过构造函数完成,Context 结构体的相关数据的初始化通过 init() 函数完成,构造函数 MD5() 和 init() 函数的实现过程与 MD4 类中对应函数的实现方法完全一致。

计算哈希值过程中的各常量的初始化见程序清单 17-22。

程序清单 17-22

```
const word32 MD5::FS[16]= {7,12,17,22,7,12,17,22,7,12,17,22,7,12,17,22};
const word32 MD5::GS[16]= {5,9,14,20,5,9,14,20,5,9,14,20,5,9,14,20};
const word32 MD5::HS[16]= {4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23};
const word32 MD5::IS[16]= {6,10,15,21,6,10,15,21,6,10,15,21,6,10,15,21};
const word32 MD5::FAC[16]= {
        0xd76aa478,0xe8c7b756,0x242070db,0xc1bdceee,
        0xf57c0faf,0x4787c62a,0xa8304613,0xfd469501,
        0x698098d8,0x8b44f7af,0xffff5bb1,0x895cd7be,
        0x6b901122,0xfd987193,0xa679438e,0x49b40821};
const word32 MD5::GAC[16]= {
        0xf61e2562,0xc040b340,0x265e5a51,0xe9b6c7aa,
        0xd62f105d,0x2441453,0xd8a1e681,0xe7d3fbc8,
        0x21e1cde6,0xc33707d6,0xf4d50d87,0x455a14ed,
        0xa9e3e905,0xfcefa3f8,0x676f02d9,0x8d2a4c8a};
const word32 MD5::HAC[16]= {
        0xfffa3942,0x8771f681,0x6d9d6122,0xfde5380c,
        0xa4beea44,0x4bdecfa9,0xf6bb4b60,0xbebfbc70,
        0x289b7ec6,0xeaa127fa,0xd4ef3085,0x4881d05,
        0xd9d4d039,0xe6db99e5,0x1fa27cf8,0xc4ac5665};
const word32 MD5::IAC[16]= {
        0xf4292244,0x432aff97,0xab9423a7,0xfc93a039,
        0x655b59c3,0x8f0ccc92,0xffeff47d,0x85845dd1,
        0x6fa87e4f,0xfe2ce6e0,0xa3014314,0x4e0811a1,
        0xf7537e82,0xbd3af235,0x2ad7d2bb,0xeb86d391};
```

各常量各自对应 FF(),GG(),HH() 和 II() 函数。

17.5.3 辅助函数的实现

MD5 算法中的辅助函数大致与 MD4 算法中的辅助函数相同,MD5 算法中的 encode() 函数、decode() 函数、rotateLeft() 函数与 MD4 算法中对应函数的实现方法完全相同。

MD5 算法中的 F(),G(),H() 与 MD4 算法中对应函数的实现方法类似,并增加了 I() 函数,F() 函数的详细实现代码与 MD4 算法中的 F() 函数实现代码相似,只需要将函数名称

修改为 word32 MD5∷F(word32 x,word32 y,word32 z),具体可参考程序清单 17-7。

G()函数的详细实现代码见程序清单 17-23。

程序清单 17-23

```
word32 MD5::G(word32 x,word32 y,word32 z)
{
    return (((x) & (z))|((y) & (~ z)));
}
```

G()函数的参数和返回类型与 MD4 中的 G()函数一致,但两个函数的计算方法有所不同,在 MD4 的 G()函数的返回值是(((x) & (y))|((x)& (z))|((y) & (z))),而 MD5 的 G()函数的返回值是(((x) & (z))|((y) & (~z)))。

MD5 算法中的 H()函数的实现方法与 MD4 中的 H()函数的实现方法相似,只需要将函数名称修改为 word32 MD5∷H(word32 x,word32 y,word32 z)即可,具体实现代码见程序清单 17-10。

在 MD5 算法中增加了 I()函数,I()的详细代码见程序清单 17-24。

程序清单 17-24

```
word32 MD5::I(word32 x,word32 y,word32 z)
{
    return ((y) ^ ((x) | (~ z)));
}
```

除了上述四个辅助函数之外,MD5 算法另外使用了四个辅助函数,这四个辅助函数为 FF(),GG(),HH()和 II()函数,这四个函数与 MD4 算法中的函数类似,但在运算方法和使用的常量上有所不同。

FF()函数的详细代码见程序清单 17-25。

程序清单 17-25

```
void MD5::FF(word32 &a,word32 b,word32 c,word32 d,word32 x,
            const word32 s,const word32 ac)
{
    a+=F(b,c,d)+x+ac;
    a=rotateLeft(a,s);
    a+=b;
}
```

MD5 算法的 FF()函数的参数与 MD4 算法的 FF()函数相比增加了一个参数,该参数在计算 a 中使用,MD5 算法的 FF()函数的计算过程与 MD 算法的 FF()函数的计算过程也有所不同,在循环左移后,输出的 a 值还需要加上 b。在计算过程中使用的循环左移函数与 MD4 算法中的循环左移函数完全一样,循环左移所使用的参数 s 就是常量 FS,在计算过程中所使用的 ac 就是常量 FAC。

GG()函数的详细代码见程序清单 17-26。

程序清单 17-26

```
void MD5::GG(word32 &a,word32 b,word32 c,word32 d,word32 x,
            const word32 s,const word32 ac)
{
      a+=G(b,c,d)+x+ac;
      a=rotateLeft(a,s);
      a+=b;
}
```

GG()函数的参数含义与FF()函数的参数含义相同，具体计算方法与FF()函数类似。

HH()函数的详细代码见程序清单17-27。

程序清单 17-27

```
void MD5::HH(word32 &a,word32 b,word32 c,word32 d,word32 x,
            const word32 s,const word32 ac)
{
      a+=H(b,c,d)+x+ac;
      a=rotateLeft(a,s);
      a+=b;
}
```

HH()函数的参数含义也与FF()函数的参数含义相同，具体计算方法也与FF()函数类似。

II()函数的详细代码见程序清单17-28。

程序清单 17-28

```
void MD5::II(word32 &a,word32 b,word32 c,word32 d,word32 x,
            const word32 s,const word32 ac)
{
      a+=I(b,c,d)+x+ac;
      a=rotateLeft(a,s);
      a+=b;
}
```

II()函数的参数含义也与FF()函数的参数含义相同，具体计算方法也与FF()函数类似。

17.5.4 哈希值计算过程的实现

MD5类中的transformHash()函数是哈希值计算过程的核心函数，transformHash()函数完成哈希值计算的基本过程，函数的详细实现代码见程序清单17-29。

程序清单 17-29

```
void MD5::transformHash(word32 * state,unsigned char * block)
{
      word32 A,B,C,D;
```

```
    word32 x[16];
    A=state[0];
    B=state[1];
    C=state[2];
    D=state[3];
    decode(x,block,64);
    word32 temp;
    word32 i;
    for(i=0;i<16;i++)
    {
        FF(A,B,C,D,x[i],FS[i],FAC[i]);
        temp=D;
        D=C;
        C=B;
        B=A;
        A=temp;
    }
    for(i=0;i<16;i++)
    {
        GG(A,B,C,D,x[(1+i*5)%16],GS[i],GAC[i]);
        temp=D;
        D=C;
        C=B;
        B=A;
        A=temp;
    }
    for(i=0;i<16;i++)
    {
        HH(A,B,C,D,x[(5+i*3)%16],HS[i],HAC[i]);
        temp=D;
        D=C;
        C=B;
        B=A;
        A=temp;
    }
    for(i=0;i<16;i++)
    {
        II(A,B,C,D,x[(i*7)%16],IS[i],IAC[i]);
        temp=D;
        D=C;
        C=B;
        B=A;
        A=temp;
    }
    state[0]+=A;
```

```
49        state[1]+=B;
50        state[2]+=C;
51        state[3]+=D;
52    }
```

transformHash()函数的实现方法与 MD4 中的 transformHash()函数的实现方法类似，但 MD5 中的 transformHash()函数在实现过程中共分为 4 轮，每轮进行 16 次操作，例如，代码行的第 12 行到第 20 行为第 1 轮，共进行 16 次 FF()运算，其他轮的运算和操作方式类似。在计算过程中需注意变量 x 的索引变化，在 FF()轮的计算中，x 索引的起始值为 0，每次操作后递增 1，即 x[i]。而在 GG()轮的计算中，x 索引的起始值为 1，每次递增 5，然后进行模 16 运算，得到相应的索引值，即 x[(1+5 * i)%16]。对于 H 轮为 x[(5+3 * i)%16]，对于 i 轮为 x[(7 * i)%16]。

在哈希值的计算过程中还使用了其他一些 MD4 算法中相似的相关函数，如 update()函数、final()函数、setBuffer()函数和 setMem()函数，这些函数的实现方法与 MD4()算法中对应的函数的实现方法完全一样，可以参考 MD4 算法中的具体实现。

17.5.5 测试与输出

程序的驱动通过函数 hashProcss()进行，通过 display()函数将计算结果显示出来，在具体执行过程中，只需要通过 main()函数调用 hashProcss()函数就可以完成测试。hashProcss()和 display()函数的实现方法与 MD4 算法中的 hashProcss()和 display()函数的实现方法完全相同。具体测试过程通过 main()函数执行，main()函数的详细代码见程序清单 17-30。

程序清单 17-30

```
int main()
{
    MD5 md5;
    md5.hashProcss("");
    md5.hashProcss("a");
    md5.hashProcss("abc");
    md5.hashProcss("abcdefghijklmnopqrstuvwxyz");
    md5.hashProcss("12345678901234567890123456789012345678901234567890123");
    md5.hashProcss("The quick brown fox jumps over the lazy dog");
    md5.hashProcss("The quick brown fox jumps over the lazy dog.");
    return 0;
}
```

测试数据部分选用了文献[68]和文献[69]中的测试数据，测试结果与文献中给出的结果一致，具体测试结果如下：

```
hash()=d41d8cd98f0b24e980998ecf8427e
hash(a)=cc175b9c0f1b6a831c399e269772661
hash(abc)= 90150983cd24fb0d6963f7d28e17f72
hash(abcdefghijklmnopqrstuvwxyz)=c3fcd3d76192e407dfb496cca67e13b
```

hash(12345678901234567890123456789012345678901234567890123456789012345678901234567890123)= c454a386ca4af0c7568ce51e423b2ef7

hash(The quick brown fox jumps over the lazy dog)= 9e107d9d372bb6826bd81d3542a419d6

hash(The quick brown fox jumps over the lazy dog.)= e4d99c290d0fb1ca068ffaddf22cbd0

在最后两个测试用例中，输入的字符串仅在最后的“.”上有所区别，其他内容完全相同，而计算得到的哈希值则完全不同。

17.6 习题与实践题

17.6.1 习题

1. 简要说明散列函数所需具备的基本性质。
2. 简要说明散列算法使用的基本方法。
3. 简要说明 MD4 散列算法的基本结构。
4. 简要说明 MD4 散列算法消息填充的基本原理。
5. 简要说明 MD4 散列算法长度填充的基本原理。
6. 简要说明 MD4 散列算法中各辅助函数的计算方法。
7. 简要说明 MD5 散列算法与 MD4 散列算法之间的区别。

17.6.2 实践题

1. 编程实现 MD4 散列算法，要求：需计算散列值的消息从文件读取，计算得到的散列值存储到文件。

2. 编程实现 MD5 散列算法，要求：需计算散列值的消息从文件读取，计算得到的散列值存储到文件。

第18章

SHA-1算法

SHA 是安全散列算法(Secure Hash Algorithm)的简称,SHA 是由美国国家安全局设计的系列算法,该系列算法共有 5 个算法,分别为 SHA-1、SHA-224、SHA-256、SHA-384 和 SHA-512,SHA-1 是该系列的算法之一,主要应用于数字签名标准(DSS)中定义的数字签名(DSA)算法。

18.1 SHA-1 算法原理

SHA-1 算法在一定程度上模仿了 MD5 算法。SHA-1 算法是以任意长度的消息文件作为输入,产生一个 160 位的消息摘要作为输出,在计算消息摘要过程中,消息是以 512 位进行分组处理。

SHA-1 计算哈希值过程中的每步操作的基本结构如图 18-1 所示。

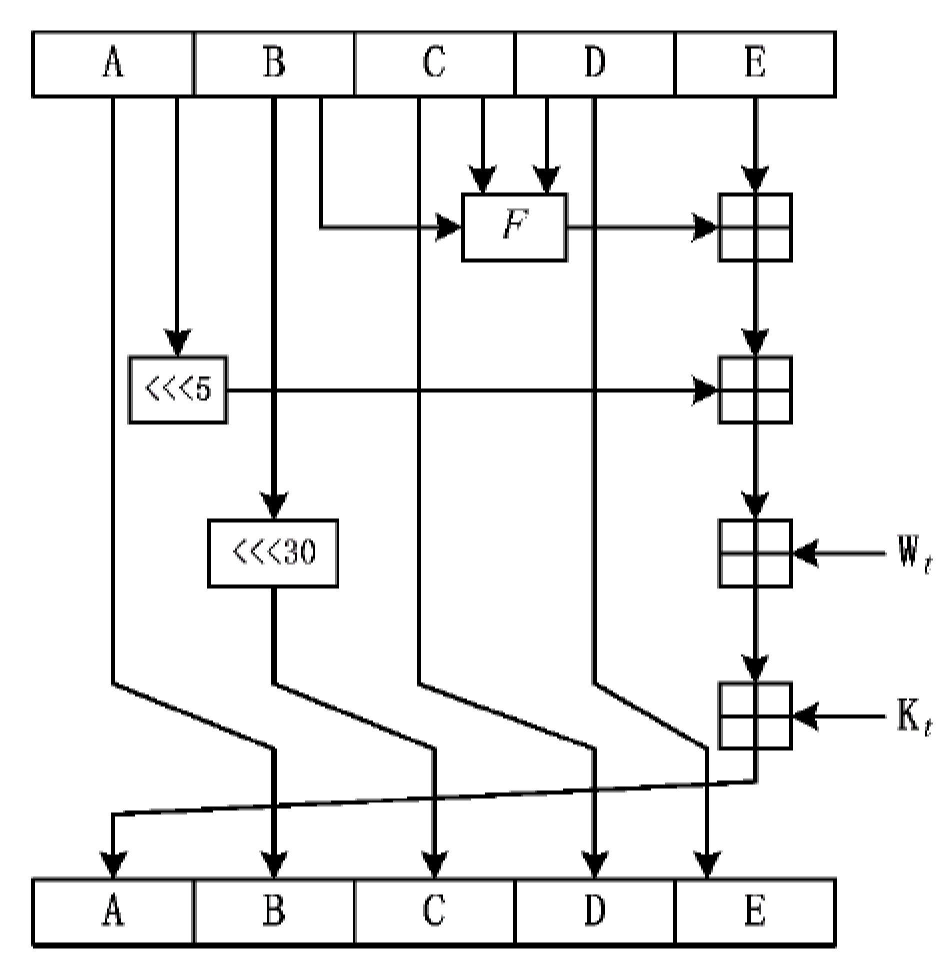


图 18-1　SHA-1 操作过程的基本结构

SHA-1 算法基本操作的输入是 5 个 32 位的字,图 18-1 中的 F 是非线性运算,W_t 是 t 轮的消息扩展的字,K_t 是 t 轮常量。

SHA-1 算法的基本过程与 MD4 和 MD5 算法类似,只是处理的位数不同,SHA-1 算法的基本计算过程如图 18-2 所示。

SHA-1 算法由以下过程完成:

(1) 消息填充、长度填充

SHA-1 算法的消息填充和长度填充的方法与 MD5 算法的填充方法完全相同,具体可参考 MD5 算法中的处理方法。

(2) 初始化 MD 缓冲区

SHA-1 算法的输出是 160 位,SHA-1 使用的链接变量缓冲区也是 160 位,用于保存连接变量和最终的哈希值。链接变量缓冲区以字(word)形式处理,共包含 5 个字,以十六进制表示,这 5 个字为:

```
A= 0x67452301
B= 0xEFCDAB89
```

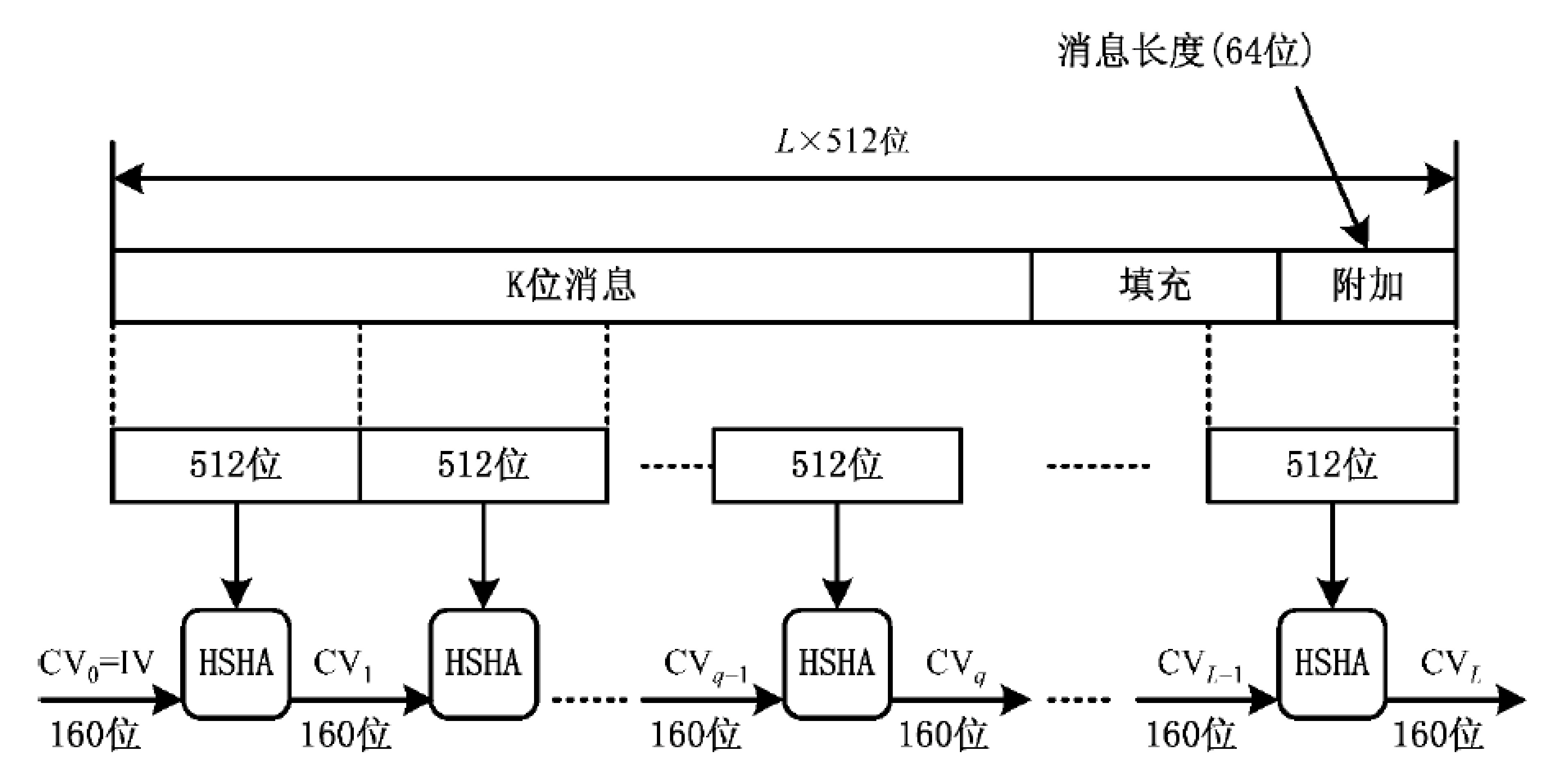


图 18-2　SHA-1 算法的基本过程

```
C= 0x98BADCFE
D= 0x10325476
E= 0xC3D2E1F0
```

A,B,C,D 这四个字的初始值与 MD5 算法的初始值一致。SHA-1 算法每次通过 5 个字 160 位进行计算,因此,与 MD5 算法相比增加了 E。

(3) 以 16 个字(word)的方式处理消息

SHA-1 算法的核心是 4 轮运算模块,这 4 轮运算的基本结构相同,但是,在每轮运算过程中使用的是不同的逻辑函数,这 4 个函数分别可以用 f_1,f_2,f_3 和 f_4 表示。这 4 个模块的处理过程如图 18-3 所示。

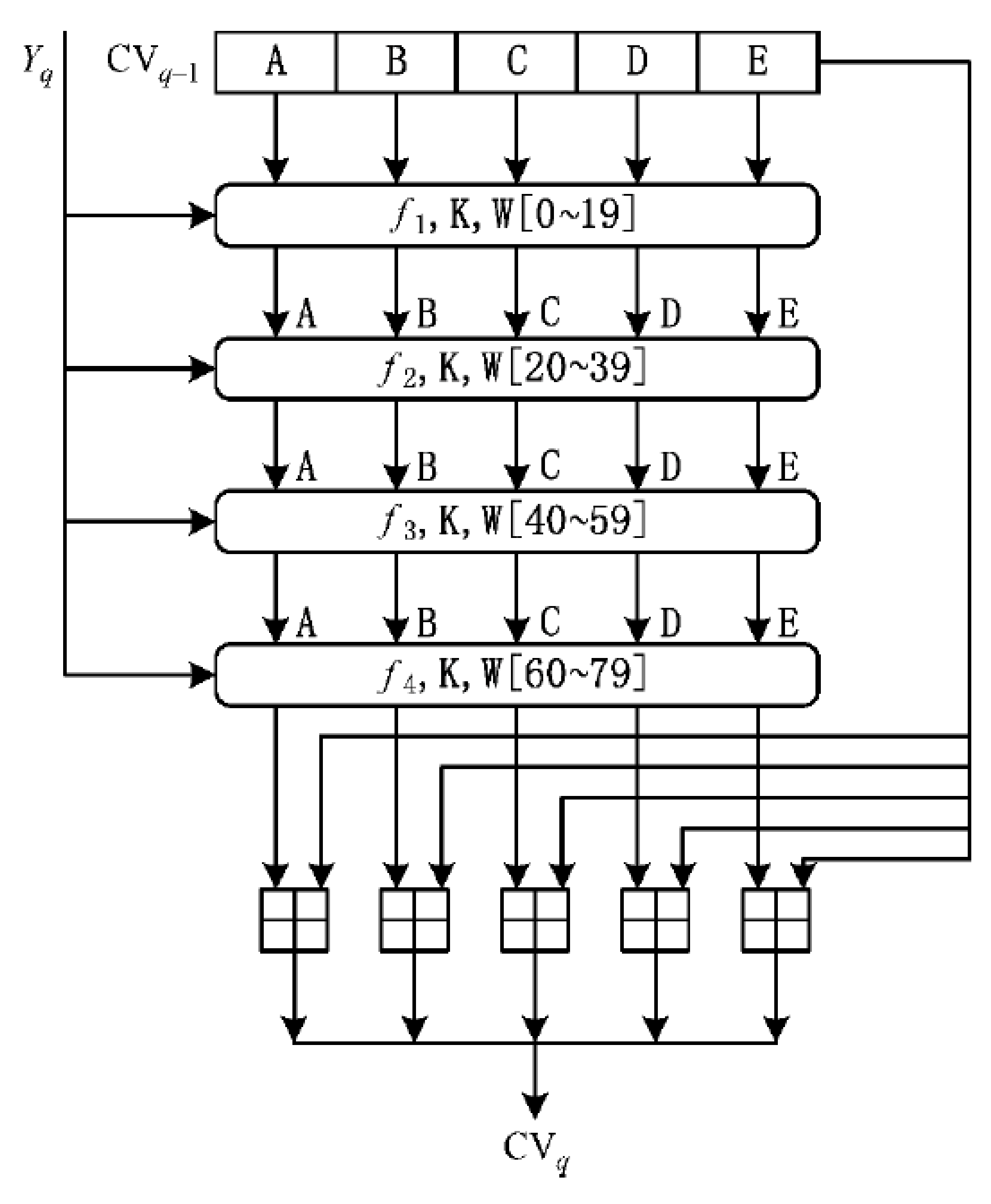


图 18-3　SHA-1 分组处理过程

每轮输入是当前要处理的512位分组和160位缓冲区ABCDE，每轮运算都要对缓冲区进行更新。在每轮运算过程中还要用到加法常量K，在不同的轮次中K的取值不同，具体取值方法如下：

t(步)	K(十六进制)	含义(取整数部分)
$0\leqslant t\leqslant 19$	0x5A827999	$2^{30}\times 2^{1/2}$
$20\leqslant t\leqslant 39$	0x6ED9EBA1	$2^{30}\times 3^{1/2}$
$40\leqslant t\leqslant 59$	0x8F1BBCDC	$2^{30}\times 5^{1/2}$
$60\leqslant t\leqslant 79$	0xCA62C1D6	$2^{30}\times 10^{1/2}$

最后一轮的输出与第一轮的输入相加得到输出。

在每一轮中每步运算的基本过程见图18-1，具体每步的计算方式如下：

$$E+f(t,B,C,D)+S^5(A)+W_t+K_t\rightarrow A$$
$$A\rightarrow B$$
$$S^{30}(B)\rightarrow C$$
$$C\rightarrow D$$
$$D\rightarrow E$$

$f(t,B,C,D)$运算根据操作步数而定，具体运算方法如下：

$0\leqslant t\leqslant 19$　$f(t,B,C,D)=(A\ \text{AND}\ C)\ \text{OR}\ ((\text{NOT}\ B)\ \text{AND}\ D)$

$20\leqslant t\leqslant 39$　$f(t,B,C,D)=B\ \text{XOR}\ C\ \text{XOR}\ D$

$40\leqslant t\leqslant 59$　$f(t,B,C,D)=(B\ \text{AND}\ C)\ \text{OR}\ (B\ \text{AND}\ D)\ \text{OR}\ (C\ \text{AND}\ D)$

$60\leqslant t\leqslant 79$　$f(t,B,C,D)=B\ \text{XOR}\ C\ \text{XOR}\ D$

S运算表示循环左移运算，例如：$S^5(A)$表示将A循环左移5位。

W_t的运算方法如下：

$$W_t=S^1(W_{t-16}\ \text{XOR}\ W_{t-14}\ \text{XOR}\ W_{t-8}\ \text{XOR}\ W_{t-3})$$

W_t是从512位分组消息中导出，具体导出方法如图18-4所示。

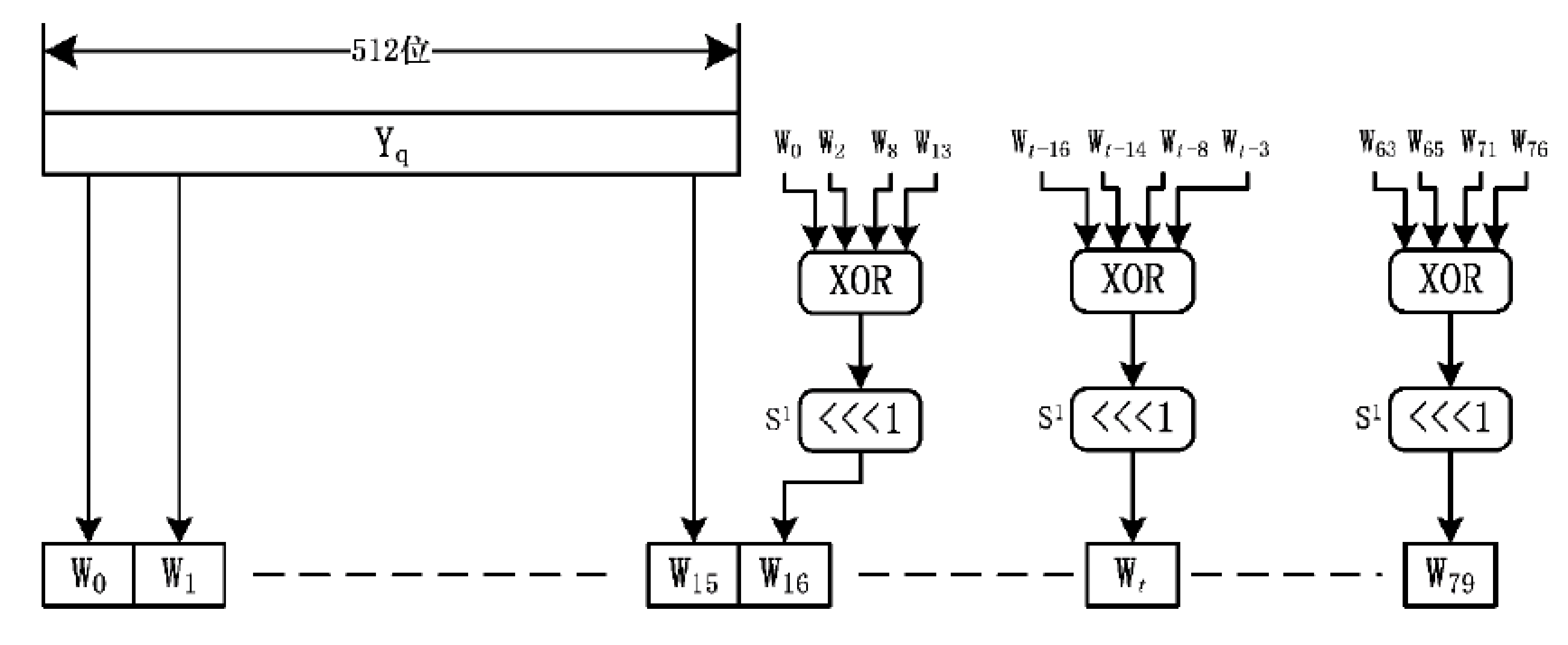


图18-4　SHA-1中W_t的处理方式

W_t的处理过程中的前16个字直接等于消息分组的第t个字，其余的字按照W_t的计算方式获得。W_t的计算也是SHA-1与MD5等算法的重要区别之一。

(4) 输出

在对所有的L个512位分组计算完毕之后，第L个分组的输出就是160位的消息摘要。

18.2 SHA-1 算法实现

SHA-1 算法实现的基本内容包括数据初始化、数据填充、计算哈希值等几部分组成，基本数据和功能都封装在 SHA_1 类中。

18.2.1 SHA-1 算法实现的基本结构

SHA-1 算法的处理单元仍然是 32 位的数据，数据通过 unsigned int 型数据进行处理，在处理过程中依然使用 word32 来表示 unsigned int 型数据。word32 型数据的声明如下：

```
typedef unsigned int word32;
```

计算哈希值过程中使用的基本数据仍使用结构体来进行处理，结构体 Context 的具体声明如下：

```
struct Context
{
    word32 low;
    word32 high;
    word32 hash[5];
    unsigned char buffer[64];
    int bufferIndex;
    int computed;
};
```

在结构体 Context 中，low 和 high 用于处理输入的长度，hash[5]是用来处理计算哈希值的缓冲，buffer[64]是用来存储计算 512 位数据的输入，bufferIndex 是用来记录读取数据的具体位置，computed 是用来记录当前数据是否已经计算。

完成具体计算的各个函数和所需的成员变量均封装在 SHA_1 类中，SHA_1 类的基本结构如图 18-5 所示。

SHA_1	
- wt[80]	: word32
- Kt[4]	: static const word32
- messageDigest[20]	: unsigned char
+ reset (Context * context)	: void
+ initW (unsigned char * input)	: void
+ leftShift (word32 in, int n)	: word32
+ transformHash (Context * context)	: void
+ padMessage (Context * context)	: void
+ input (Context * context, const unsigned char * messageInput, word32 length)	: void
+ result (Context * context, unsigned char * digest)	: void
+ test (char * newInput)	: void
+ display ()	: void

图 18-5　SHA_1 类的基本结构

SHA_1 类的具体声明见程序清单 18-1。

程序清单 18-1

```
class SHA_1
{
    public:
        void reset(Context * context);
        void initW(unsigned char * input);
        word32 leftShift(word32 in,int n);
        void transformHash(Context * context);
        void padMessage(Context * context);
        void input(Context * context,const unsigned char * messageInput,word32 length);
        void result(Context * context,unsigned char * digest);
        void test(char * newInput);
        void display();
    private:
        word32 wt[80];
        static const word32 Kt[4];
        unsigned char messageDigest[20];
};
```

在程序清单 18-1 中，变量 wt[80]用于存放 t 次运算消息扩展的字，变量 Kt[4]用于存放在 4 轮运算过程中所需的常量，而 messageDigest[20]则用于存放消息摘要。

SHA_1 类中各成员函数的作用如下：

- reset()——用于重置结构体 Context 中的各成员变量。
- initW()——用于初始化消息扩展的字。
- leftShift()——实现 32 位字的循环左移。
- transformHash()——用于计算哈希值。
- padMessage()——用于进行消息填充。
- input()——用于获取输入，即获得需要计算哈希值的字符串。
- result()——完成哈希值的计算，并将哈希值转换为字符型数据作为输出。
- test()——用于驱动测试过程。
- display()——用于显示计算得到的哈希值。

18.2.2 数据初始化

SHA_1 类的数据初始化工作由 reset()，initW()和 input()等若干函数来实现，同时还需要初始化常量 Kt。各函数在计算过程的不同阶段实现对不同数据初始化，其作用各不相同。reset()函数主要用于初始化 Context 结构体中的各成员变量，reset()函数的详细代码见程序清单 18-2。

程序清单 18-2

```
void SHA_1::reset(Context * context)
{
```

```
    context->low=0;
    context->high=0;
    context->bufferIndex=0;
    context->hash[0]=0x67452301;
    context->hash[1]=0xEFCDAB89;
    context->hash[2]=0x98BADCFE;
    context->hash[3]=0x10325476;
    context->hash[4]=0xC3D2E1F0;
    context->computed=0;
}
```

reset()函数在初始化过程中并没有初始化 Context 结构体中的 buffer[64]数组，该数组在计算哈希值过程中进行初始化，最终由输入的数据、填充的数据和附加的数据长度确定。计算哈希值的缓冲与 MD4 与 MD5 算法相比，增加了第 5 个数据缓冲。

在计算输入消息的哈希值之前，还需要初始化 wt，wt 数据初始化实际上是由 buffer[64]数组确定，这部分的功能通过函数 initW()实现，initW()函数的详细代码见程序清单 18-3。

程序清单 18-3

```
void SHA_1::initW(unsigned char * input)
{
    word32 i;
    for(i=0;i<16;i++)
    {
        wt[i]=input[i * 4]<<24;
        wt[i]|=input[i * 4+1]<<16;
        wt[i]|=input[i * 4+2]<<8;
        wt[i]|=input[i * 4+3];
    }
    for(i=16;i<80;i++)
    {
        wt[i]=leftShift((wt[i-3]^wt[i-8]^wt[i-14]^wt[i-16]),1);
    }
}
```

wt 初始化分为两部分，一部分是直接将输入的 64 个 unsigned char 数据直接转化为 32 位的 word 型数据，转化的具体方法如图 18-6 所示。

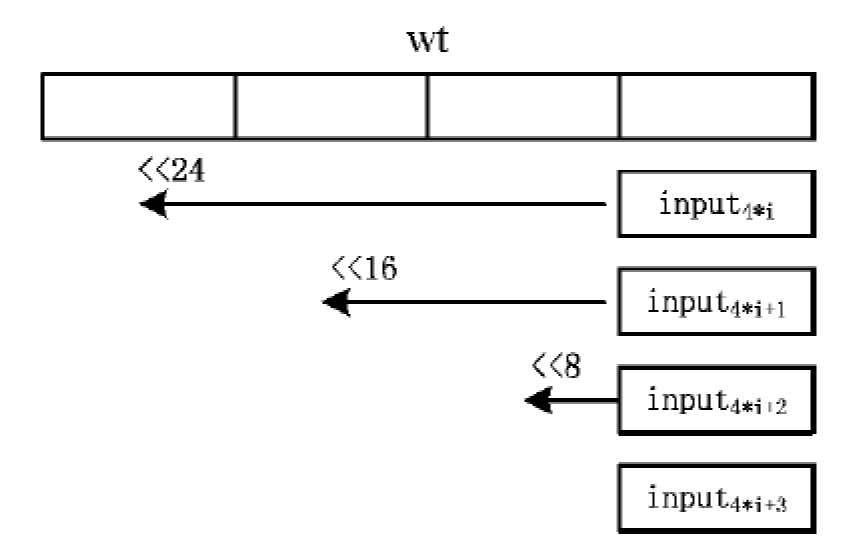


图 18-6　wt 移位运算过程示意图

每次运算过程是将 4 个 unsigned char 型数据转换为 1 个 word32 型数据，共进行 16 次运算，获得 wt[0]～wt[15]，wt[16]～wt[79]是通过 wt[0]～wt[15]的对应数据进行“异或”运算，并将计算得到的结果进行循环左移后获得。在计算 wt[16]～wt[79]的过程中使用了循环左移函数，循环左移函数的详细

代码见程序清单 18-4。

程序清单 18-4

```
word32 SHA_1::leftShift(word32 in,int n)
{
      return (in<<n)|(in>>32-n);
}
```

input()函数在计算过程中用于初始化,input()将在计算哈希值过程中进行介绍。

在初始化过程中还需要对常量 Kt 进行初始化,常量 Kt 的初始化见程序清单 18-5。

程序清单 18-5

```
const word32 SHA_1::Kt[4]={
                     0x5A827999,0x6ED9EBA1,0x8F1BBCDC,0xCA62C1D6};
```

18.2.3 哈希值计算过程的实现

哈希值的计算过程为:处理输入数据,当数据达到 64 个字节(512)时就计算哈希值,若数据长度达不到 64 字节时,就进入最后处理,最后处理过程包括数据填充和附加输入数据的长度,然后计算哈希值。这些功能分别通过函数 transformHash(),padMessage(),input()和 result()实现。

transformHash()函数是计算哈希值的核心函数,该函数共包含 4 轮 80 次运算,通过 4 轮 80 次运算完成一组数据的计算,transformHash()函数的详细代码见程序清单 18-6。

程序清单 18-6

```
void SHA_1::transformHash(Context * context)
{
      word32 A,B,C,D,E;
      word32 i;
      word32 temp;
      initW(context->buffer);
      A=context->hash[0];
      B=context->hash[1];
      C=context->hash[2];
      D=context->hash[3];
      E=context->hash[4];
      for(i=0;i<20;i++)
      {
            temp=leftShift(A,5)+((B&C)|((~B)&D))+E+wt[i]+Kt[0];
            E=D;
            D=C;
            C=leftShift(B,30);
            B=A;
            A=temp;
      }
```

```
    for(i=20;i<40;i++)
    {
        temp=leftShift(A,5)+(B^C^D)+E+wt[i]+Kt[1];
        E=D;
        D=C;
        C=leftShift(B,30);
        B=A;
        A=temp;
    }
    for(i=40;i<60;i++)
    {
        temp=leftShift(A,5)+((B&C)|(B&D)|(C&D))+E+wt[i]+Kt[2];
        E=D;
        D=C;
        C=leftShift(B,30);
        B=A;
        A=temp;
    }
    for(i=60;i<80;i++)
    {
        temp=leftShift(A,5)+(B^C^D)+E+wt[i]+Kt[3];
        E=D;
        D=C;
        C=leftShift(B,30);
        B=A;
        A=temp;
    }
    context->hash[0]+=A;
    context->hash[1]+=B;
    context->hash[2]+=C;
    context->hash[3]+=D;
    context->hash[4]+=E;
    context->bufferIndex=0;
}
```

transformHash()函数的参数为 Context 结构体，transformHash()函数计算哈希值共分为 3 个部分，首先将 Context 结构体的哈希值数据缓冲赋给临时变量 A，B，C，D 和 E，然后经过 4 轮 80 次运算，最后计算得到 Context 结构体的哈希值。在每一轮的计算过程中，均经过 20 次运算，每轮运算方法与常量各不相同。在计算过程中使用了循环左移运算，该运算过程与初始化过程中的循环左移实现方法相同。

在哈希值计算过程中还需用到数据填充和附加函数，数据填充和附加通过函数 padMessage()实现，padMessage()的详细代码见程序清单 18-7。

程序清单 18-7

```
void SHA_1::padMessage(Context * context)
```

```
02 {
03     if(context->bufferIndex>55)
04     {
05         context->buffer[context->bufferIndex++]=0x80;
06         while(context->bufferIndex<64)
07         {
08             context->buffer[context->bufferIndex++]=0;
09         }
10         transformHash(context);
11         while(context->bufferIndex<56)
12         {
13             context->buffer[context->bufferIndex++]=0;
14         }
15     }
16     else
17     {
18         context->buffer[context->bufferIndex++]=0x80;
19         while(context->bufferIndex<56)
20         {
21             context->buffer[context->bufferIndex++]=0;
22         }
23     }
24     context->buffer[56]=context->high>>24;
25     context->buffer[57]=context->high>>16;
26     context->buffer[58]=context->high>>8;
27     context->buffer[59]=context->high;
28     context->buffer[60]=context->low>>24;
29     context->buffer[61]=context->low>>16;
30     context->buffer[62]=context->low>>8;
31     context->buffer[63]=context->low;
32     transformHash(context);
33 }
```

消息填充和消息附加的过程是根据 Context 结构体中的 buffer 的长度来确定，当填充完毕之后就由 transformHash()函数计算哈希值，更新 Context 结构体，直到全部处理完毕。SHA_1 算法的输入数据是必须经过填充的，当输入数据的长度小于 56 字节时，先将消息填充到 56 字节长度，然后填充数据长度，计算哈希值，否则先将当前分组填充到 64 字节，计算哈希值，然后再将下一分组填充到 56 字节，之后附加消息长度，计算哈希值。第 3 行代码到第 15 行代码用于处理输入数据的长度大于 55 字节的情况，第 16 行代码到第 22 行代码是处理输入数据小于或等于 55 字节的情况。程序清单中的第 23 行到第 30 行代码是附加输入数据的长度，在附加数据长度时需要将 32 位的 word32 型数据转换为 8 位的 unsigned char 型数据，其方法是每次取出其中的 8 位数据。

input()函数的功能是读取待计算哈希值的消息，input()函数的详细代码见程序清单 18-8。

程序清单 18-8

```
01 void SHA_1::input(Context * context,const unsigned char * messageInput,word32 length)
02 {
03
04     while(length--)
05     {
06         context->buffer[context->bufferIndex++]= * messageInput&0xFF;
07         context->low+=8;
08         if(context->low==0)
09         {
10             context->high++;
11         }
12         if(context->bufferIndex==64)
13         {
14             transformHash(context);
15         }
16         messageInput++;
17     }
18 }
```

input()函数的参数是 Context 结构体、输入的消息和消息的长度。input()函数的处理过程为：当消息还没有处理完成时，读入消息，并将消息存储到 Context 结构体的 buffer 数组中，当 Context 结构体的 buffer 数组长度达到 64(512 位)时，则计算哈希值。

在 input()函数中，第 7 行代码的作用是每读入一个字节，则总数的低位加 8，这是字节与位区别形成的结果，第 8 行代码到第 11 行代码是处理进位问题，第 12 行代码到第 15 行代码的作用是每读入 64 字节的数据则计算一次哈希值。

result()函数的功能是处理最后的计算结果，并将 word32 型消息摘要转换为 unsigned char 型消息摘要，result()函数的详细代码见程序清单 18-9。

程序清单 18-9

```
void SHA_1::result(Context * context,unsigned char * digest)
{
    int i;
    if(!context->computed)
    {
        padMessage(context);
        for(i=0;i<64;i++)
        {
            context->buffer[i]=0;
        }
        context->low=0;
        context->high=0;
        context->computed=1;
    }
```

```
15      for(i=0;i<20;i++)
16      {
17          digest[i]=context->hash[i>>2]>>8*(3-(i&0x03));
18      }
19  }
```

result()函数首先判断最后部分消息是否已经计算哈希值，若没有计算，则填充并计算哈希值，计算完成后，将哈希值的数据缓冲转换为 unsigned char 型消息摘要，转换过程见代码行的第 15 行到第 18 行。在代码行第 17 行中，i>>2 相当于 i/4，而>>8*(3－(i&0x03))的作用是先取高 8 位，并逐步向低位运行，每次取 8 位，例如，当 i=0 时，相当于 hash[0]>>24，当 i=1 时，相当于 hash[0]>>16，以此类推。

18.2.4　测试与输出

测试和输出通过函数 test()和 display()来完成，由主函数 main()来驱动。

test()函数的详细代码见程序清单 18-10。

程序清单 18-10

```
void SHA_1::test(char * newInput)
{
    cout<<"message(\""<<newInput<<"\")"<<endl;
    Context context;
    word32 length;
    length=strlen(newInput);
    reset(&context);
    input(&context,(const unsigned char *)newInput,length);
    result(&context,messageDigest);
    display();
}
```

test()函数首先调用 reset()函数，对 Context 结构体进行初始化操作，然后读取输入消息并计算消息摘要，最后将消息摘要输出。消息摘要由函数 display()完成，display()函数的详细代码见程序清单 18-11。

程序清单 18-11

```
void SHA_1::display()
{
    word32 i;
    cout<<"digest(";
    for(i=0;i<20;i++)
    {
        cout<<hex<<int(messageDigest[i])<<" ";
    }
    cout<<")"<<endl;
}
```

display()函数的功能仅是将消息摘要以 16 进制的形式输出。

main()函数最终执行整个测试的驱动，main()函数的详细代码见程序清单 18-12。

程序清单 18-12

```
int main()
{
    SHA_1 sha;
    sha.test("The quick brown fox jumps over the lazy dog");
    sha.test("The quick brown fox jumps over the lazy cog");
    sha.test("");
    sha.test("abc");
    return 0;
}
```

main()函数的测试用例选用的是参考文献[70]和[71]中的测试用例，具体测试结果如下：

```
message("The quick brown fox jumps over the lazy dog")
digest(2f d4 e1 c6 7a 2d 28 fc ed 84 9e e1 bb 76 e7 39 1b 93 eb 12)
message("The quick brown fox jumps over the lazy cog")
digest(de 9f 2c 7f d2 5e 1b 3a fa d3 e8 5a b d1 7d 9b 10 d b4 b3)
message("")
digest(da 39 a3 ee 5e 6b 4b d 32 55 bf ef 95 60 18 90 af d8 7 9)
message("abc")
digest(a9 99 3e 36 47 6 81 6a ba 3e 25 71 78 50 c2 6c 9c d0 d8 9d)
```

测试结果与文献中的测试结果一致。

在前两项的测试用例中，仅将最后的“dog”改为“cog”，而计算得到的结果则完全不同。

18.3 习题与实践题

18.3.1 习题

1. 简要说明 SHA-1 计算哈希值过程中的每步操作的基本结构(见图 18-1)，并说明计算方法。

2. 简要说明 SHA-1 算法的 4 轮运算模块的运算原理。

18.3.2 实践题

参考 18.2 节的 SHA-1 算法的实现过程，编程实现 SHA-1 算法，要求：待计算哈希值的数据从相应的文件读取，计算得到的哈希值存储到相应的文件。

第19章

RIPEMD-160算法

RIPEMD 散列算法是一系列散列算法，是欧洲 RIPE(RACE Integrity Primitives Evaluation)计划下在 Leuven Belgium(鲁汶，比利时)由一组研究人员设计的。最初该算法是 128 位长度的，在经过改造后成为 160 位的。

19.1 RIPEMD-160 算法原理

RIPEMD-160 算法的输入长度可以是任意位的，以 512 位的分组进行处理，输出的是 160 位(5 个 32 位的字)组成的消息摘要。RIPEMD-160 算法是在大量研究 MD4 和 MD5 算法基础上设计的，算法的基本过程也与 MD5 算法类似。RIPEMD-160 算法实施的基本过程如下：

(1) 消息填充

对需要计算消息摘要的消息进行填充，使填充后的长度满足：长度≡448 mod 512，若长度已经满足上述条件，仍然需要进行填充，即填充 512 位，填充数据的长度在 1 位至 512 位之间，填充的第 1 位为“1”，其他位为“0”。

(2) 长度填充

经过填充后的消息长度与分组长度相差 64 位，该 64 位填充的内容为：L mod 2^{64}，其中 L 为消息的长度。

(3) 初始化 MD 缓冲区

RIPEMD-160 算法的消息摘要存储在 160 位的缓冲区中，该缓冲区用 5 个 32 位的寄存器 A,B,C,D 和 E 表示，这 5 个值与 SHA-1 算法的 5 个值相同，分别为

```
A= 0x67452301
B= 0xEFCDAB89
C= 0x98BADCFE
D= 0x10325476
E= 0xC3D2E1F0
```

(4) 轮运算

这步是以 16 个字(512 位)为分组进行的运算，是 RIPEMD-160 算法的核心。10 轮运算共分为 2 组，每组 5 轮，每轮执行 16 步运算，具体运算过程如图 19-1 所示。

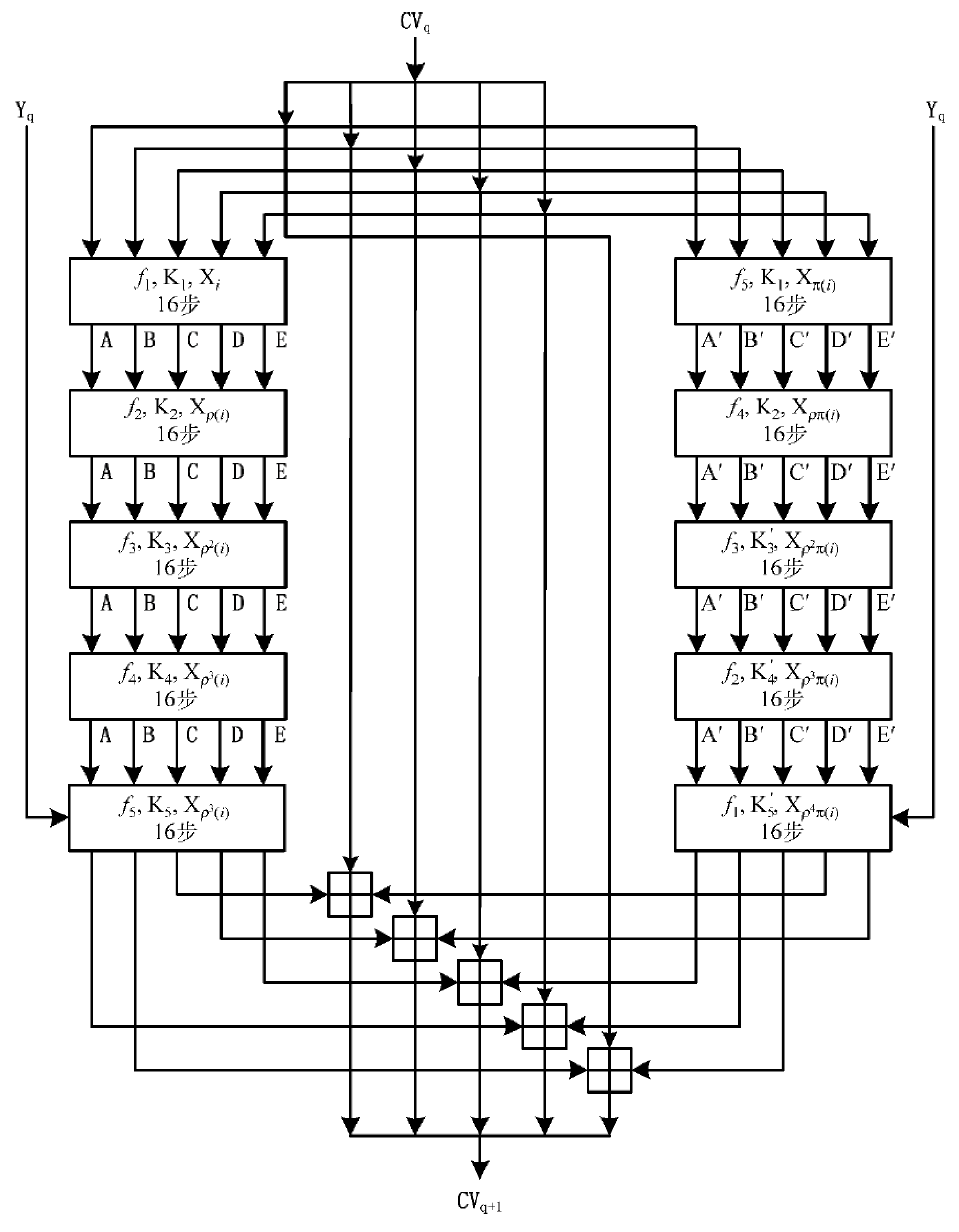


图 19-1　RIPEMD-160 算法分组运算过程

运算结果可以表示如下：

$$
\begin{aligned}
CV_{q+1}[0] &= CV_q[1]+C+D' \\
CV_{q+1}[1] &= CV_q[2]+D+E' \\
CV_{q+1}[2] &= CV_q[3]+E+A' \\
CV_{q+1}[3] &= CV_q[4]+A+B' \\
CV_{q+1}[4] &= CV_q[0]+B+C'
\end{aligned}
\qquad (19\text{-}1)
$$

RIPEMD-160 算法的分组运算分为左、右两部分，每部分又分为 5 轮运算，每轮运算又分为 16 步，每轮运算中使用的基本逻辑函数是 f_1, f_2, f_3, f_4, f_5，使用逻辑函数的顺序左右两半正好相反。

各逻辑函数的运算方式如下：

$$
\begin{aligned}
f_1(x,y,z) &= x \oplus y \oplus z \\
f_2(x,y,z) &= (x \wedge y) \vee (\neg x \wedge z) \\
f_3(x,y,z) &= (x \vee \neg y) \oplus z \\
f_4(x,y,z) &= (x \wedge z) \vee (y \wedge \neg z) \\
f_5(x,y,z) &= x \oplus (y \vee \neg z)
\end{aligned}
\tag{19-2}
$$

各 f 函数在各轮中的使用由表 19-1 给出。

表 19-1　不同轮次的 f 常量使用

	第 1 轮	第 2 轮	第 3 轮	第 4 轮	第 5 轮
左半部分	f_1	f_2	f_3	f_4	f_5
右半部分	f_5	f_4	f_3	f_2	f_1

在每轮运算中又分为相似的 16 步运算，每步运算将结果循环右移，每步运算的具体方法如图 19-2 所示。

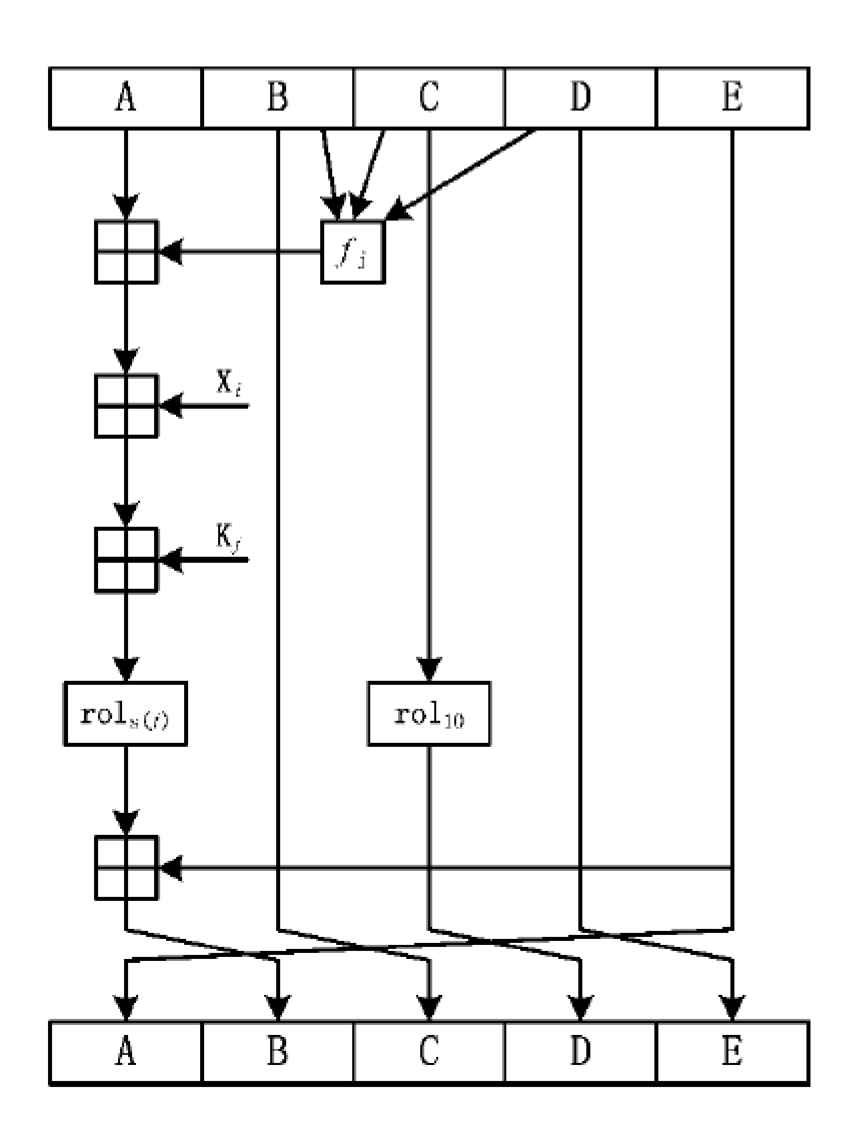


图 19-2　RIPEMD-160 每步运算过程

每步的运算过程可以表示为

$$
\begin{aligned}
&A := E \\
&B := (A + f(B,C,D) + X + K) <<< s + E \\
&C := B \\
&D := C <<< 10 \\
&E := D
\end{aligned}
\tag{19-3}
$$

在每步运算过程中使用了常量 K，不同轮次和左右各半的常量各不相同，各轮中使用的 K 常量的具体值由表 19-2 给出。

在计算过程中还用到了另一个变量 X，在计算过程中将每次要处理的 512 位分组以 word 格式存储到 X[0…15]数组中，各字的基本置换顺序由表 19-3 给出。

表 19-2 轮常量 K

轮 次	左半部分		右半部分	
	计算方法	十六进制(取整)	计算方法	十六进制(取整)
第 1 轮	0	0x00000000	$2^{30}\sqrt[3]{2}$	0x50A28BE6
第 2 轮	$2^{30}\sqrt{2}$	0x5A827999	$2^{30}\sqrt[3]{3}$	0x5C4DD124
第 3 轮	$2^{30}\sqrt{3}$	0x6ED9EBA1	$2^{30}\sqrt[3]{5}$	0x6D703EF3
第 4 轮	$2^{30}\sqrt{5}$	0x8F1BBCDC	$2^{30}\sqrt[3]{7}$	0x7A6D76E9
第 5 轮	$2^{30}\sqrt{7}$	0xA953FD4E	0	0x00000000

表 19-3 置换顺序

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\rho(i)$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8

设 $\pi(i)=9i+5$，那么各轮中的置换算法由表 19-4 给出。

表 19-4 各轮置换算法

	第 1 轮	第 2 轮	第 3 轮	第 4 轮	第 5 轮
左半部分	i	ρ	ρ^2	ρ^3	ρ^4
右半部分	π	$\rho\pi$	$\rho^2\pi$	$\rho^3\pi$	$\rho^4\pi$

对经过置换的字还需要进行循环左移，循环左移具体位数见表 19-5。

表 19-5 循环左移

轮	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
1	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
2	12	13	11	15	6	9	9	7	12	15	11	13	7	8	7	7
3	13	15	14	11	7	7	6	8	13	14	13	12	5	5	6	9
4	14	11	12	14	8	6	5	5	15	12	15	14	9	9	8	6
5	15	12	13	13	9	5	8	6	14	11	12	11	8	6	5	5

(5) 输出

在处理完所有分组之后，最后分组的输出就是 160 位的消息摘要。

19.2 RIPEMD-160 算法实现

RIPEMD-160 算法的实现包括数据初始化、计算哈希值、数据输出和数据测试等几部分组成，各成员变量和成员函数都封装在 RIPEMD160 类中。

19.2.1 RIPEMD-160 算法实现的基本结构

RIPEMD-160 算法的基本数据单元仍然是 32 位的数据，在 RIPEMD-160 算法的实现过程中仍然使用 word32 型数据，其声明如下：

```
typedef unsigned int word32;
```

RIPEMD-160 算法各基本功能的实现和相应的变量均封装在 RIPEMD160 类中，RIPEMD160 类的基本结构如图 19-3 所示。

RIPEMD160	
- hashBuffer[5] : word32 - hashCode[20] : unsigned char	
+ F (word32 x, word32 y, word32 z)	: word32
+ G (word32 x, word32 y, word32 z)	: word32
+ H (word32 x, word32 y, word32 z)	: word32
+ I (word32 x, word32 y, word32 z)	: word32
+ J (word32 x, word32 y, word32 z)	: word32
+ leftShift (word32 x, word32 n)	: word32
+ FF (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ GG (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ HH (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ II (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ JJ (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ FFF (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ GGG (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ HHH (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ III (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ JJJ (word32 &a, word32 b, word32 &c, word32 d, word32 e, word32 x, word32 s)	: void
+ initHashBuffer ()	: void
+ hashProcess (word32 *mdBuffer, word32 *X)	: void
+ finish (word32 *mdBuffer, unsigned char *strIn, word32 length)	: void
+ MD (unsigned char *message)	: void
+ mdDisplay ()	: void
+ byteToWord (unsigned char *str)	: word32

图 19-3 RIPEMD160 类的基本结构

RIPEMD160 类的具体声明见程序清单 19-1。

程序清单 19-1

```
class RIPEMD160
{
    public:
        word32 F(word32 x,word32 y,word32 z);
        word32 G(word32 x,word32 y,word32 z);
        word32 H(word32 x,word32 y,word32 z);
        word32 I(word32 x,word32 y,word32 z);
        word32 J(word32 x,word32 y,word32 z);
        word32 leftShift(word32 x,word32 n);
        void FF(word32 &a,word32 b,word32 &c,word32 d,word32 e,word32 x,word32 s);
        void GG(word32 &a,word32 b,word32 &c,word32 d,word32 e,word32 x,
            word32 s);
        void HH(word32 &a,word32 b,word32 &c,word32 d,word32 e,word32 x,
            word32 s);
        void II(word32 &a,word32 b,word32 &c,word32 d,word32 e,word32 x,word32 s);
        void JJ(word32 &a,word32 b,word32 &c,word32 d,word32 e,word32 x,word32 s);
        void FFF(word32 &a,word32 b,word32 &c,word32 d,
            word32 e,word32 x,word32 s);
```

```
        void GGG(word32 &a,word32 b,word32 &c,word32 d,
            word32 e,word32 x,word32 s);
        void HHH(word32 &a,word32 b,word32 &c,word32 d,
            word32 e,word32 x,word32 s);
        void III(word32 &a,word32 b,word32 &c,word32 d,
            word32 e,word32 x,word32 s);
        void JJJ(word32 &a,word32 b,word32 &c,word32 d,
            word32 e,word32 x,word32 s);
        void initHashBuffer();
        void hashProcess(word32 * mdBuffer,word32 * X);
        void finish(word32 * mdBuffer,unsigned char * strIn,word32 length);
        void MD(unsigned char * message);
        void mdDisplay();
        word32 byteToWord(unsigned char * str);
    private:
        word32 hashBuffer[5];
        unsigned char hashCode[20];
};
```

在 RIPEMD160 类中，成员变量 hashBuffer[5]适用于计算哈希值的缓冲，hashCode[20]适用于存储计算得到的哈希值。

RIPEMD160 类的各成员函数的作用如下：

- F(),G(),H(),I(),J()——对应算法中的 $f_1 \sim f_5$ 函数，执行相应的逻辑运算。
- FF(),GG(),HH(),II(),JJ()——左半部分的各轮运算。
- FFF(),GGG(),HHH(),III(),JJJ()——右半部分的各轮运算。
- leftShift()——word32 型数据的循环左移。
- initHashBuffer()——初始化计算哈希值用数据缓冲。
- hashProcess()——计算哈希值的函数。
- finish()——完成最后一轮哈希值计算。
- MD()——运行对应消息的哈希值计算。
- byteToWord()——将字节型数据转化为 word32 型数据。
- mdDisplay()——显示最终的计算结果。

19.2.2 数据初始化

数据初始化主要任务是初始化计算哈希值用的数据缓冲，数据初始化通过初始化函数 initHashBuffer()完成，initHashBuffer()函数的详细代码见程序清单 19-2。

程序清单 19-2

```
void RIPEMD160::initHashBuffer()
{
    hashBuffer[0]=0x67452301;
    hashBuffer[1]=0xEFCDAB89;
    hashBuffer[2]=0x98BADCFE;
    hashBuffer[3]=0x10325476;
```

```
    hashBuffer[4]=0xC3D2E1F0;
}
```

RIPEMD-160 算法的数据缓冲初值与 SHA-1 算法的数据缓冲的初值相同。

19.2.3 辅助函数的实现

RIPEMD-160 算法的辅助函数包括基本的逻辑运算函数、左半部分的计算函数和右半部分的计算函数组成。基本的逻辑函数包括 F()函数、G()函数、H()函数、I()函数和 J()函数。

F()函数的实现代码见程序清单 19-3。

程序清单 19-3

```
word32 RIPEMD160::F(word32 x,word32 y,word32 z)
{
    return x^y^z;
}
```

G()函数的详细代码见程序清单 19-4。

程序清单 19-4

```
word32 RIPEMD160::G(word32 x,word32 y,word32 z)
{
    return (x&y)|(~x&z);
}
```

H()函数的详细代码见程序清单 19-5。

程序清单 19-5

```
word32 RIPEMD160::H(word32 x,word32 y,word32 z)
{
    return (x|~y)^z;
}
```

I()函数的详细代码见程序清单 19-6。

程序清单 19-6

```
word32 RIPEMD160::I(word32 x,word32 y,word32 z)
{
    return (x&z)|(y&~z);
}
```

J()函数的详细代码见程序清单 19-7。

程序清单 19-7

```
word32 RIPEMD160::J(word32 x,word32 y,word32 z)
{
    return x^(y|~z);
}
```

以上 5 个函数实现 RIPEMD160 算法的 5 个基本逻辑运算。

左半部分各轮的基本计算过程通过函数 FF()、GG()、HH()、II()和 JJ()函数完成。

FF()函数的详细代码见程序清单 19-8。

程序清单 19-8

```
void RIPEMD160::FF(word32 &a,word32 b,word32 &c,word32 d,
                        word32 e,word32 x,word32 s)
{
      a+=F(b,c,d)+x;
      a=leftShift(a,s)+e;
      c=leftShift(c,10);
}
```

FF()函数在运算过程中使用了 F()函数，F()函数的参数为计算哈希值的数据缓冲，参数 x 是以 word32 形式表示的输入的消息，参数 s 是用于循环移位的参数，即循环左移的大小。在第 1 轮运算过程中由于常量 K=0，因此，在函数实现过程中没有出现常量 K。

循环左移函数 leftShift()的详细代码见程序清单 19-9。

程序清单 19-9

```
word32 RIPEMD160::leftShift(word32 x,word32 n)
{
      return (x<<n)|x>>(32-n);
}
```

GG()函数的实现方法与 FF()函数的实现方法类似，函数参数也相同。GG()函数的详细代码见程序清单 19-10。

程序清单 19-10

```
void RIPEMD160::GG(word32 &a,word32 b,word32 &c,word32 d,
                  word32 e,word32 x,word32 s)
{
      a+=G(b,c,d)+x+0x5A827999;
      a=leftShift(a,s)+e;
      c=leftShift(c,10);
}
```

与 FF()函数相比，GG()函数出现了 0x5A827999，该参数为常量 K。其他部分与 FF()函数基本相同。

HH()函数的详细代码见程序清单 19-11。

程序清单 19-11

```
void RIPEMD160::HH(word32 &a,word32 b,word32 &c,word32 d,
                  word32 e,word32 x,word32 s)
{
      a+=H(b,c,d)+x+0x6ED9EBA1;
```

```
    a= leftShift(a,s)+e;
    c= leftShift(c,10);
}
```

II()函数的详细代码见程序清单 19-12。

程序清单 19-12

```
void RIPEMD160::II(word32 &a,word32 b,word32 &c,word32 d,
                   word32 e,word32 x,word32 s)
{
    a+=I(b,c,d)+x+0x8F1BBCDC;
    a= leftShift(a,s)+e;
    c= leftShift(c,10);
}
```

JJ()函数的详细代码见程序清单 19-13。

程序清单 19-13

```
void RIPEMD160::JJ(word32 &a,word32 b,word32 &c,word32 d,
                   word32 e,word32 x,word32 s)
{
    a+=J(b,c,d)+x+0xA953FD4E;
    a= leftShift(a,s)+e;
    c= leftShift(c,10);
}
```

右半部分各轮的基本计算过程通过函数 FFF(),GGG(),HHH(),III()和 JJJ()完成，只是在使用过程中与左半部分的使用顺序正好相反。各函数与左半部分各函数的实现方法类似。

FFF()函数的详细代码见程序清单 19-14。

程序清单 19-14

```
void RIPEMD160::FFF(word32 &a,word32 b,word32 &c,word32 d,
                   word32 e,word32 x,word32 s)
{
    a+=F(b,c,d)+x;
    a= leftShift(a,s)+e;
    c= leftShift(c,10);
}
```

GGG()函数的详细代码见程序清单 19-15。

程序清单 19-15

```
void RIPEMD160::GGG(word32 &a,word32 b,word32 &c,word32 d,
                   word32 e,word32 x,word32 s)
{
    a+=G(b,c,d)+x+0x7A6D76E9;
```

```
    a=leftShift(a,s)+e;
    c=leftShift(c,10);
}
```

HHH()函数的详细代码见程序清单 19-16。

程序清单 19-16

```
void RIPEMD160::HHH(word32 &a,word32 b,word32 &c,word32 d,
                word32 e,word32 x,word32 s)
{
    a+=H(b,c,d)+x+0x6D703EF3;
    a=leftShift(a,s)+e;
    c=leftShift(c,10);
}
```

III()函数的详细代码见程序清单 19-17。

程序清单 19-17

```
void RIPEMD160::III(word32 &a,word32 b,word32 &c,word32 d,
                word32 e,word32 x,word32 s)
{
    a+=I(b,c,d)+x+0x5C4DD124;
    a=leftShift(a,s)+e;
    c=leftShift(c,10);
}
```

JJJ()函数的详细代码见程序清单 19-18。

程序清单 19-18

```
void RIPEMD160::JJJ(word32 &a,word32 b,word32 &c,word32 d,
                word32 e,word32 x,word32 s)
{
    a+=J(b,c,d)+x+0x50A28BE6;
    a=leftShift(a,s)+e;
    c=leftShift(c,10);
}
```

以上各函数将在具体计算消息的哈希值时使用。

19.2.4 哈希值计算过程的实现

哈希值计算过程的核心函数是 hashProcess(),它完成一组消息的哈希值计算,该函数的参数是哈希值缓冲和需计算哈希值的消息分组。

hashProcess()函数的详细代码见程序清单 19-19。

程序清单 19-19

```
void RIPEMD160::hashProcess(word32 * mdBuffer,word32 * X)
```

```
{
    word32 aL,bL,cL,dL,eL;
    word32 aR,bR,cR,dR,eR;
    aL=mdBuffer[0];
    bL=mdBuffer[1];
    cL=mdBuffer[2];
    dL=mdBuffer[3];
    eL=mdBuffer[4];
    aR=mdBuffer[0];
    bR=mdBuffer[1];
    cR=mdBuffer[2];
    dR=mdBuffer[3];
    eR=mdBuffer[4];
    FF(aL,bL,cL,dL,eL,X[0],11);
    FF(eL,aL,bL,cL,dL,X[1],14);
    FF(dL,eL,aL,bL,cL,X[2],15);
    FF(cL,dL,eL,aL,bL,X[3],12);
    FF(bL,cL,dL,eL,aL,X[4],5);
    FF(aL,bL,cL,dL,eL,X[5],8);
    FF(eL,aL,bL,cL,dL,X[6],7);
    FF(dL,eL,aL,bL,cL,X[7],9);
    FF(cL,dL,eL,aL,bL,X[8],11);
    FF(bL,cL,dL,eL,aL,X[9],13);
    FF(aL,bL,cL,dL,eL,X[10],14);
    FF(eL,aL,bL,cL,dL,X[11],15);
    FF(dL,eL,aL,bL,cL,X[12],6);
    FF(cL,dL,eL,aL,bL,X[13],7);
    FF(bL,cL,dL,eL,aL,X[14],9);
    FF(aL,bL,cL,dL,eL,X[15],8);
    GG(eL,aL,bL,cL,dL,X[7],7);
    GG(dL,eL,aL,bL,cL,X[4],6);
    GG(cL,dL,eL,aL,bL,X[13],8);
    GG(bL,cL,dL,eL,aL,X[1],13);
    GG(aL,bL,cL,dL,eL,X[10],11);
    GG(eL,aL,bL,cL,dL,X[6],9);
    GG(dL,eL,aL,bL,cL,X[15],7);
    GG(cL,dL,eL,aL,bL,X[3],15);
    GG(bL,cL,dL,eL,aL,X[12],7);
    GG(aL,bL,cL,dL,eL,X[0],12);
    GG(eL,aL,bL,cL,dL,X[9],15);
    GG(dL,eL,aL,bL,cL,X[5],9);
    GG(cL,dL,eL,aL,bL,X[2],11);
    GG(bL,cL,dL,eL,aL,X[14],7);
    GG(aL,bL,cL,dL,eL,X[11],13);
    GG(eL,aL,bL,cL,dL,X[8],12);
```

```
HH(dL,eL,aL,bL,cL,X[3],11);
HH(cL,dL,eL,aL,bL,X[10],13);
HH(bL,cL,dL,eL,aL,X[14],6);
HH(aL,bL,cL,dL,eL,X[4],7);
HH(eL,aL,bL,cL,dL,X[9],14);
HH(dL,eL,aL,bL,cL,X[15],9);
HH(cL,dL,eL,aL,bL,X[8],13);
HH(bL,cL,dL,eL,aL,X[1],15);
HH(aL,bL,cL,dL,eL,X[2],14);
HH(eL,aL,bL,cL,dL,X[7],8);
HH(dL,eL,aL,bL,cL,X[0],13);
HH(cL,dL,eL,aL,bL,X[6],6);
HH(bL,cL,dL,eL,aL,X[13],5);
HH(aL,bL,cL,dL,eL,X[11],12);
HH(eL,aL,bL,cL,dL,X[5],7);
HH(dL,eL,aL,bL,cL,X[12],5);
II(cL,dL,eL,aL,bL,X[1],11);
II(bL,cL,dL,eL,aL,X[9],12);
II(aL,bL,cL,dL,eL,X[11],14);
II(eL,aL,bL,cL,dL,X[10],15);
II(dL,eL,aL,bL,cL,X[0],14);
II(cL,dL,eL,aL,bL,X[8],15);
II(bL,cL,dL,eL,aL,X[12],9);
II(aL,bL,cL,dL,eL,X[4],8);
II(eL,aL,bL,cL,dL,X[13],9);
II(dL,eL,aL,bL,cL,X[3],14);
II(cL,dL,eL,aL,bL,X[7],5);
II(bL,cL,dL,eL,aL,X[15],6);
II(aL,bL,cL,dL,eL,X[14],8);
II(eL,aL,bL,cL,dL,X[5],6);
II(dL,eL,aL,bL,cL,X[6],5);
II(cL,dL,eL,aL,bL,X[2],12);
JJ(bL,cL,dL,eL,aL,X[4],9);
JJ(aL,bL,cL,dL,eL,X[0],15);
JJ(eL,aL,bL,cL,dL,X[5],5);
JJ(dL,eL,aL,bL,cL,X[9],11);
JJ(cL,dL,eL,aL,bL,X[7],6);
JJ(bL,cL,dL,eL,aL,X[12],8);
JJ(aL,bL,cL,dL,eL,X[2],13);
JJ(eL,aL,bL,cL,dL,X[10],12);
JJ(dL,eL,aL,bL,cL,X[14],5);
JJ(cL,dL,eL,aL,bL,X[1],12);
JJ(bL,cL,dL,eL,aL,X[3],13);
JJ(aL,bL,cL,dL,eL,X[8],14);
JJ(eL,aL,bL,cL,dL,X[11],11);
```

```
JJ(dL,eL,aL,bL,cL,X[6],8);
JJ(cL,dL,eL,aL,bL,X[15],5);
JJ(bL,cL,dL,eL,aL,X[13],6);
JJJ(aR,bR,cR,dR,eR,X[5],8);
JJJ(eR,aR,bR,cR,dR,X[14],9);
JJJ(dR,eR,aR,bR,cR,X[7],9);
JJJ(cR,dR,eR,aR,bR,X[0],11);
JJJ(bR,cR,dR,eR,aR,X[9],13);
JJJ(aR,bR,cR,dR,eR,X[2],15);
JJJ(eR,aR,bR,cR,dR,X[11],15);
JJJ(dR,eR,aR,bR,cR,X[4],5);
JJJ(cR,dR,eR,aR,bR,X[13],7);
JJJ(bR,cR,dR,eR,aR,X[6],7);
JJJ(aR,bR,cR,dR,eR,X[15],8);
JJJ(eR,aR,bR,cR,dR,X[8],11);
JJJ(dR,eR,aR,bR,cR,X[1],14);
JJJ(cR,dR,eR,aR,bR,X[10],14);
JJJ(bR,cR,dR,eR,aR,X[3],12);
JJJ(aR,bR,cR,dR,eR,X[12],6);
III(eR,aR,bR,cR,dR,X[6],9);
III(dR,eR,aR,bR,cR,X[11],13);
III(cR,dR,eR,aR,bR,X[3],15);
III(bR,cR,dR,eR,aR,X[7],7);
III(aR,bR,cR,dR,eR,X[0],12);
III(eR,aR,bR,cR,dR,X[13],8);
III(dR,eR,aR,bR,cR,X[5],9);
III(cR,dR,eR,aR,bR,X[10],11);
III(bR,cR,dR,eR,aR,X[14],7);
III(aR,bR,cR,dR,eR,X[15],7);
III(eR,aR,bR,cR,dR,X[8],12);
III(dR,eR,aR,bR,cR,X[12],7);
III(cR,dR,eR,aR,bR,X[4],6);
III(bR,cR,dR,eR,aR,X[9],15);
III(aR,bR,cR,dR,eR,X[1],13);
III(eR,aR,bR,cR,dR,X[2],11);
HHH(dR,eR,aR,bR,cR,X[15],9);
HHH(cR,dR,eR,aR,bR,X[5],7);
HHH(bR,cR,dR,eR,aR,X[1],15);
HHH(aR,bR,cR,dR,eR,X[3],11);
HHH(eR,aR,bR,cR,dR,X[7],8);
HHH(dR,eR,aR,bR,cR,X[14],6);
HHH(cR,dR,eR,aR,bR,X[6],6);
HHH(bR,cR,dR,eR,aR,X[9],14);
HHH(aR,bR,cR,dR,eR,X[11],12);
HHH(eR,aR,bR,cR,dR,X[8],13);
```

```
    HHH(dR,eR,aR,bR,cR,X[12],5);
    HHH(cR,dR,eR,aR,bR,X[2],14);
    HHH(bR,cR,dR,eR,aR,X[10],13);
    HHH(aR,bR,cR,dR,eR,X[0],13);
    HHH(eR,aR,bR,cR,dR,X[4],7);
    HHH(dR,eR,aR,bR,cR,X[13],5);
    GGG(cR,dR,eR,aR,bR,X[8],15);
    GGG(bR,cR,dR,eR,aR,X[6],5);
    GGG(aR,bR,cR,dR,eR,X[4],8);
    GGG(eR,aR,bR,cR,dR,X[1],11);
    GGG(dR,eR,aR,bR,cR,X[3],14);
    GGG(cR,dR,eR,aR,bR,X[11],14);
    GGG(bR,cR,dR,eR,aR,X[15],6);
    GGG(aR,bR,cR,dR,eR,X[0],14);
    GGG(eR,aR,bR,cR,dR,X[5],6);
    GGG(dR,eR,aR,bR,cR,X[12],9);
    GGG(cR,dR,eR,aR,bR,X[2],12);
    GGG(bR,cR,dR,eR,aR,X[13],9);
    GGG(aR,bR,cR,dR,eR,X[9],12);
    GGG(eR,aR,bR,cR,dR,X[7],5);
    GGG(dR,eR,aR,bR,cR,X[10],15);
    GGG(cR,dR,eR,aR,bR,X[14],8);
    FFF(bR,cR,dR,eR,aR,X[12],8);
    FFF(aR,bR,cR,dR,eR,X[15],5);
    FFF(eR,aR,bR,cR,dR,X[10],12);
    FFF(dR,eR,aR,bR,cR,X[4],9);
    FFF(cR,dR,eR,aR,bR,X[1],12);
    FFF(bR,cR,dR,eR,aR,X[5],5);
    FFF(aR,bR,cR,dR,eR,X[8],14);
    FFF(eR,aR,bR,cR,dR,X[7],6);
    FFF(dR,eR,aR,bR,cR,X[6],8);
    FFF(cR,dR,eR,aR,bR,X[2],13);
    FFF(bR,cR,dR,eR,aR,X[13],6);
    FFF(aR,bR,cR,dR,eR,X[14],5);
    FFF(eR,aR,bR,cR,dR,X[0],15);
    FFF(dR,eR,aR,bR,cR,X[3],13);
    FFF(cR,dR,eR,aR,bR,X[9],11);
    FFF(bR,cR,dR,eR,aR,X[11],11);
    dR += cL + hashBuffer[1];
    hashBuffer[1]=hashBuffer[2]+ dL+ eR;
    hashBuffer[2]=hashBuffer[3]+ eL+ aR;
    hashBuffer[3]=hashBuffer[4]+ aL+ bR;
    hashBuffer[4]=hashBuffer[0]+ bL+ cR;
    hashBuffer[0]=dR;
}
```

hashProcess()函数共由 4 部分组成：初始化左右两部分计算哈希值的初始值，这部分通过代码行第 3 行到第 14 行实现；实现左半部分的哈希值计算，这部分通过代码行第 15 行到第 94 行完成；实现右半部分哈希值计算，这部分通过代码行第 95 行到第 174 行完成；最后部分将计算结果合并获得最终结果。在左右各半的计算过程中，每部分又分成 5 轮，每轮 16 步计算。

19.2.5 测试与输出

测试与输出部分由 MD()函数、finish()函数和 display()函数等组成，最终通过主函数进行驱动。

执行哈希值计算的函数 MD()的具体代码见程序清单 19-20。

程序清单 19-20

```
void RIPEMD160::MD(unsigned char * message)
{
    word32 i;
    word32 length;
    word32 nBytes;
    word32 X[16];
    initHashBuffer();
    length=strlen((char*)message);
    for(nBytes=length;nBytes>63;nBytes-=64)
    {
        for(i=0;i<16;i++)
        {
            X[i]=byteToWord(message);
            message+=4;
        }
        hashProcess(hashBuffer,X);
    }
    finish(hashBuffer,message,length);
    for(i=0;i<20;i+=4)
    {
        hashCode[i]=hashBuffer[i>>2];
        hashCode[i+1]=(hashBuffer[i>>2]>>8);
        hashCode[i+2]=(hashBuffer[i>>2]>>16);
        hashCode[i+3]=(hashBuffer[i>>2]>>24);
    }
    cout<<"hash(";
    for(i=0;i<length;i++)
    {
        cout<<message[i];
    }
    cout<<")=";
    mdDisplay();
```

```
}
```

MD()函数的参数是 unsigned char 型指针，具体输入需要计算哈希值的消息。对于够 64 字节的消息则直接计算消息哈希值，并将计算结果存储到计算哈希值的缓冲，若不够 64 字节则调用 finish()函数，完成最后的数据填充和哈希值计算。

在计算过程中使用函数 byteToWord()将 unsigned char 型数据转换为 word32 型数据，函数的详细代码见程序清单 19-21。

程序清单 19-21

```
word32 RIPEMD160::byteToWord(unsigned char * str)
{
    return ((word32) * ((str)+ 3)<< 24)|((word32) * ((str)+ 2)<< 16)|
        ((word32) * ((str)+ 1)<< 8)|((word32) * (str));
}
```

将 unsigned char 型数据转换为 word32 型数据通过移位过程来完成。

finish()函数的详细代码见程序清单 19-22。

程序清单 19-22

```
void RIPEMD160::finish(word32 * mdBuffer,unsigned char * strIn,word32 length)
{
    word32 i;
    word32 X[16];
    for(i= 0;i< 16;i+ + )
    {
        X[i]= 0;
    }
    for(i= 0;i< (length&63);i+ + )
    {
        X[i>> 2]^= (word32) * strIn+ + << (8 * (i&3));
    }
    X[(length>> 2)&15]^= (word32)1<< (8 * (length&3)+ 7);
    if((length&63)> 55)
    {
        hashProcess(mdBuffer,X);
    }
    X[14]= length<< 3;
    X[15]= (length>> 29);
    hashProcess(mdBuffer,X);
}
```

在哈希值计算完成之后，通过 MD()函数 32 位 word32 型数据转换为 unsigned char 型数据，并将最终结果通过 mdDisplay()函数输出，mdDisplay()函数的详细代码见程序清单 19-23。

程序清单 19-23

```
void RIPEMD160::mdDisplay()
```

```
{
    word32 i;
    for(i=0;i<20;i++)
    {
        cout<<setw(2)<<setfill('0')<<hex<<(int)hashCode[i];
    }
}
```

输出函数将计算结果以 16 进制格式输出。

测试过程的驱动通过主函数来完成，测试用例选择了参考文献[77]中的示例进行测试，主函数的具体代码见程序清单 19-24。

程序清单 19-24

```
int main()
{
    RIPEMD160 rmd;
    rmd.MD((unsigned char *)"");
    rmd.MD((unsigned char *)"a");
    rmd.MD((unsigned char *)"abc");
    rmd.MD((unsigned char *)"message digest");
    rmd.MD((unsigned char *)"abcdefghijklmnopqrstuvwxyz");
    return 0;
}
```

```
hash()=9c1185a5c5e9fc54612808977ee8f548b2258d31
hash(a)=0bdc9d2d256b3ee9daae347be6f4dc835a467ffe
hash(abc)=8eb208f7e05d987a9b044a8e98c6b087f15a0bfc
hash(message digest)=5d0689ef49d2fae572b881b123a85ffa21595f36
hash(abcdefghijklmnopqrstuvwxyz)=f71c27109c692c1b56bbdceb5b9d2865b3708dbc
```

测试结果与文献给出的结果一致。

19.3 习题与实践题

19.3.1 习题

1. 简要说明 RIPEMD-160 算法的分组运算过程的基本原理。
2. 参考图 19-2 说明 RIPEMD-160 算法每步运算过程的基本原理。

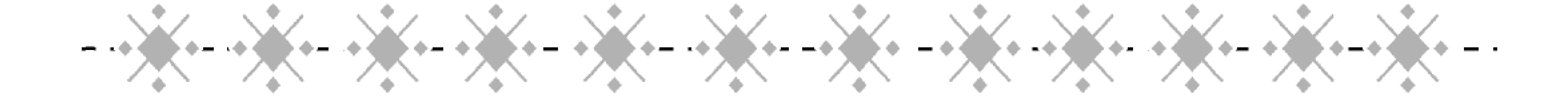
19.3.2 实践题

编程实现 RIPEMD-160 算法，要求：待计算哈希值的数据从文件读取，计算得到的哈希值也存储在对应的文件中。

第　部分

数字签名

数字签名是密码技术的一种,其主要作用是进行身份验证和完整性检验。数字签名通常是公钥密码算法与散列算法的结合产物。数字签名技术在网上银行、电子商务、电子政务和网络通信等领域有着广泛的应用。



第20章

数 字 签 名

数字签名(digital signature)最早由 Whitfield Diffie 和 Martin Hellamn 在 1976 年提出,之后逐步形成一系列签名体制,不同签名体制主要依赖于不同的数学困难问题。目前,比较常见的签名算法包括 RSA 签名算法、Rabin 签名算法、Elgamal 签名算法和椭圆曲线签名算法等。

20.1 数字签名概述

数字签名是一种类似于在纸上进行的物理签名,主要用于数字信息的鉴别。数字签名实际上就是把数字形式的消息与某个发送消息的实体相联系的数字信息,通常是将这个数字信息附加到消息后面,以便于消息的接收方能够方便地进行鉴别,并证明消息来源于真正的发送方。

数字签名一般包括两部分,一部分是签名,另一部分是验证。数字签名的基本过程如图 20-1 所示。

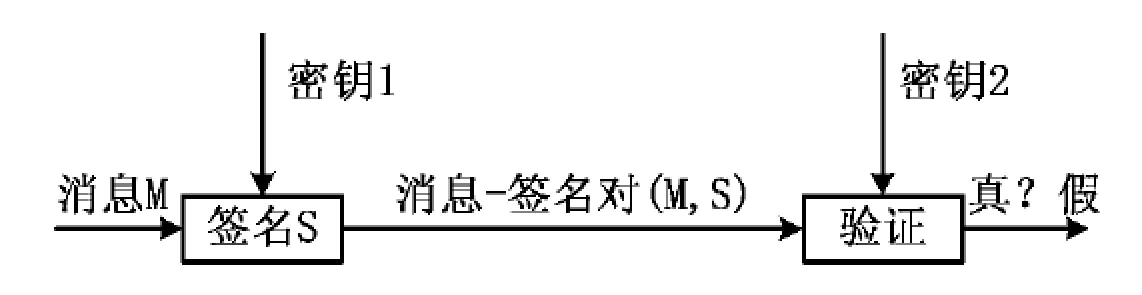


图 20-1 数字签名与验证的基本过程

数字签名算法主要包含 3 部分内容:

(1) 密钥生成算法:主要生成用于签名的私钥和用于验证的公钥。

(2) 签名算法:对于给定的消息和私钥产生签名。

(3) 验证算法:对于给定的公钥、消息和签名可以验证消息来源、消息是否被篡改等。

数字签名通常是针对待签名的哈希值进行签名,数字签名的具体过程如下:

(1) 针对所需签名的消息使用哈希函数计算哈希值。

(2) 使用私钥对哈希值进行签名。

(3) 将签名附加在消息的后面,并发送给对方。

接收者在接收到签名之后,需验证数字签名,验证签名的基本过程如下:

(1) 将接收到数据进行分割,一部分为签名,一部分为消息。

(2) 将消息部分使用哈希函数计算得到相应的哈希值。

(3) 将签名部分用签名者的公钥解密,获得签名者计算得到的哈希值。

(4) 计算得到的哈希值与解密得到的哈希值进行比较,若两者相同表示签名为真,若两者不同则签名存在问题。

图20-2是典型的数字签名过程示意图,图20-2的左方是签名者的签名过程,签名的基本过程包括计算消息的哈希值、使用私钥对哈希值进行签名并附加在消息后面发送给接收方。接收方在收到消息之后,将消息和签名部分分离,计算消息的哈希值并与解密得到的哈希值相比较,若两者相同则表示消息未被篡改,否则消息被篡改。

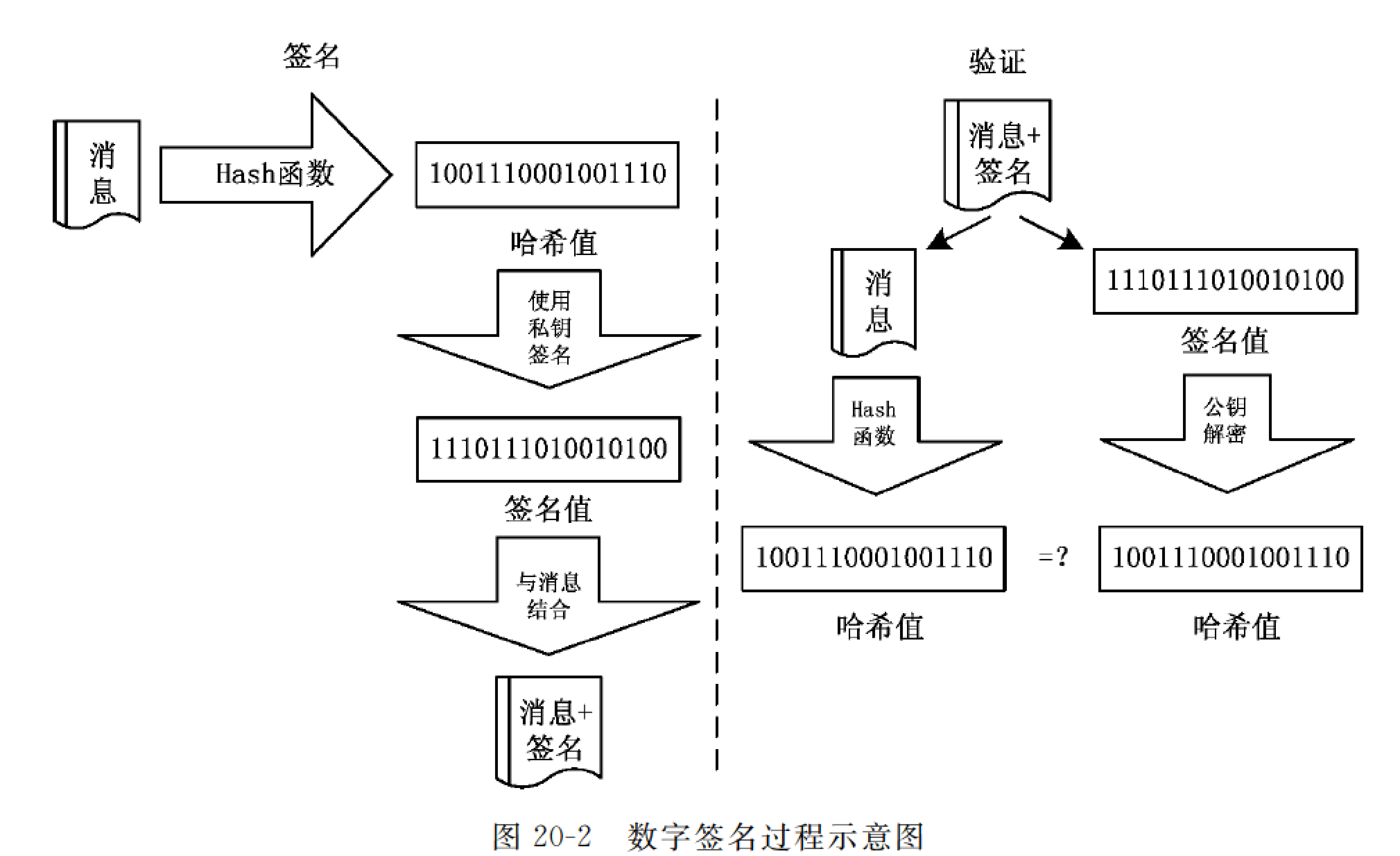


图20-2 数字签名过程示意图

20.2 RSA数字签名方案

RSA数字签名方案是使用较早的数字签名方案之一,也是由MIT的Ronal Rivest,Adi Shamir和Len Adleman提出。在2000年美国联邦信息处理标准(FIPS PUB 186)的修改方案中RSA数字签名方案成为推荐方案之一。

RSA数字签名实际上是通过RSA加密算法进行简单改造后得到的,通常由于明文M都比较长,因此,一般都不对明文进行直接签名,而是利用明文计算明文的哈希值(Hash(M)),然后再对哈希值进行签名。RSA数字签名方案在许多安全标准中得到广泛的应用,美国联邦信息处理标准(Federal Information Processing Standards)FIPS186-2也将RSA数字签名算法最为其推荐算法,ISO/IEC9796也推荐使用RSA数字签名算法作为标准数字签名算法。

RSA数字签名方案既可以直接对消息进行签名,也可以对消息的消息摘要进行签名。由于使用RSA数字签名算法对消息直接进行签名的效率太低,因此,一般情况下是对消息摘要进行签名。同时,由于签名过程使用的是私钥,而验证过程使用的是公钥,因此,对整个消息进行签名的意义不大。

RSA数字签名方案的基本实施过程如图20-3所示。

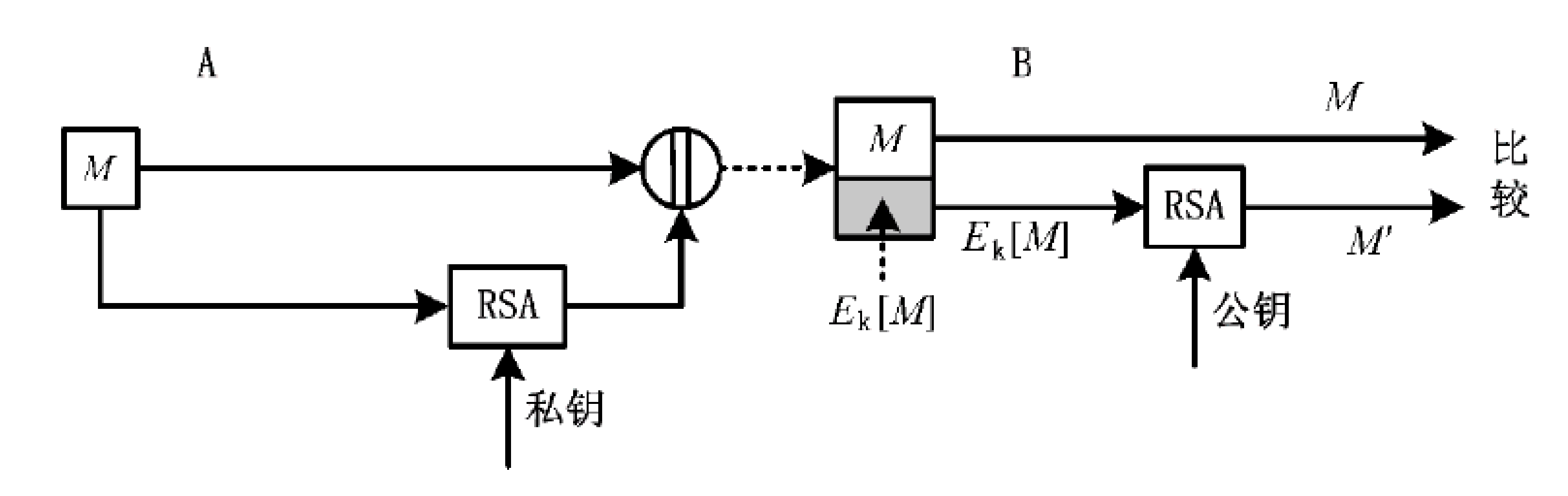


图 20-3 RSA 数字签名方案示意图

假设用户 A 的公钥为 n_A,e_A,私钥为 d_A,待签名的消息为 M(或消息摘要),那么,使用 RSA 签名方案的用户 A 签名算法为

$$s \equiv M^{d_A} \bmod n_A$$

用户 B 的验证算法为

$$M' \equiv s^{e_A} \bmod n_A$$

如果得到的 M'与 M 相同,则表示消息未被篡改,否则消息被篡改。

RSA 签名算法的签名与验证过程正好与 RSA 加密算法的加密与解密过程相反。这种只签名不加密的签名过程,任何拥有公钥的人在截获,就可以通过公钥恢复得到消息。

如果要保证信息不被泄露并进行签名,则需要进行两次运算,并需要两组密钥,一组用于加密,一组用于签名。

假设用户 A 用于签名的公钥为(n_A,e_A),私钥为(n_A,d_A),待签名的消息为 M(或消息摘要),用于加密的公钥为(n_B,e_B),私钥为(n_B,d_B),那么,用户 A 的签名与加密过程为

$$s \equiv M^{d_A} \bmod n_A$$

$$c \equiv s^{e_B} \bmod n_B$$

用户 B 在得到消息之后,先进行解密运算,然后再验证签名,具体过程为

$$s' \equiv c^{d_B} \bmod n_B$$

$$M' \equiv s'^{e_A} \bmod n_A$$

RSA 数字签名算法是基于 RSA 加密算法的安全性,由于只有签名者才知道签名密钥,因此,其他人无法伪造签名者签名。

示例 20-1 设 $p=23$,$q=7$,待签名的消息 $M=35$,试给出签名和验证的计算过程。

解 (1) $n=pq=23\times 7=161$。

(2) $\varphi(n)=\varphi(pq)=\varphi(p)\varphi(q)=(23-1)\times(7-1)=22\times 6=132$。

(3) 选择 e,有 $1<e<132$,并与 132 互素,选择 $e=2^4+1=17$,17 与 132 互素。

(4) 计算 $e(\bmod\ \varphi(n))$的乘法逆元,得到 $d=101$。

(5) 签名:$s=M^d \bmod n=35^{101} \bmod 161=98$。

(6) 验证:$M=s^e \bmod n=98^{17} \bmod 161=35$。

20.3 Elgamal 数字签名方案

Elgamal 数字签名方案由 Elgamal 公钥加密算法演变而来,最早发表于 1985 年,美国的数字签名标准(DSS)也使用了经 Elgamal 数字签名算法演变的算法。Elgamal 数字签名

的安全性是基于有限域上求解离散对数问题的困难性。

Elgamsl 数字签名方案基本实施过程如下：

(1) 密钥生成

① 生成随机大素数 p 以及 p 的本原根 g。

② 随机选择私钥 x，$1 \leqslant x \leqslant p-2$。

③ 计算 $y = g^x \bmod p$。

计算得到的公钥为(p,g,y)，私钥为 x，私钥 x 将用于签名。

(2) 签名过程

① 随即选择一整数 k，满足 $0<k<p$，并且满足 k 与 $p-1$ 互素。

② 计算 $a = g^k \bmod p$。

③ 若待签名的消息为 m，则计算

$$m = ax + ks \bmod (p-1) \tag{20-1}$$

这个方程也称为签名方程，将式(20-1)转换可以得到

$$s = k^{-1}(m - ax) \bmod (p-1) \tag{20-2}$$

得到签名(a,s)。

(3) 验证算法

要验证签名只需验证

$$y^a a^s \bmod p = g^M \bmod p \tag{20-3}$$

上述过程是直接对消息进行签名，使用这种签名方法签名速度比较慢，通常是对消息的哈希值进行签名，具体签名和验证过程如图 20-4 所示。

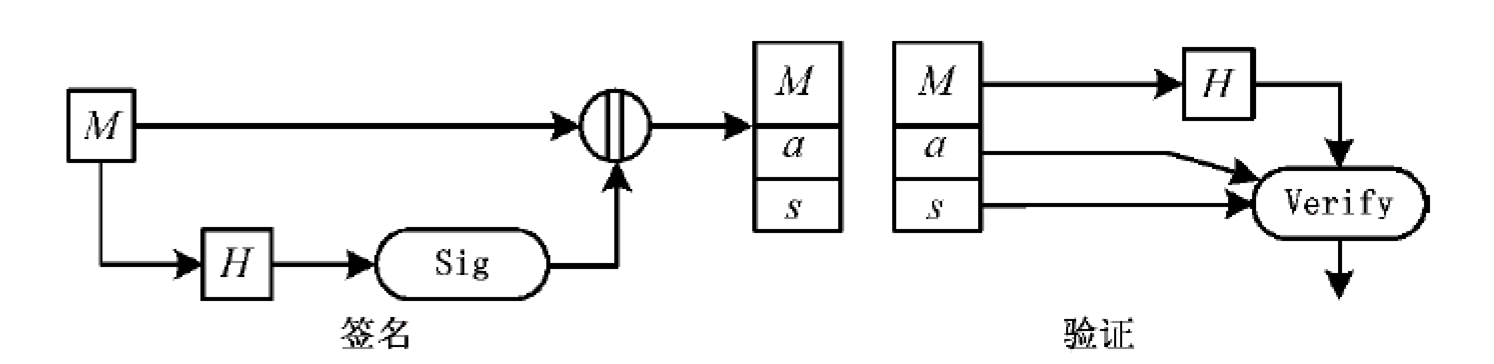


图 20-4 数字签名和验证过程

在图 20-4 描述的数字签名方案中，首先使用相应的哈希函数计算消息的哈希值，然后对哈希值进行签名，在签名之后将签名附加在消息之后发送给接收方。接收方在收到签名之后，将签名部分与消息分离，验证获得哈希值，并计算消息的哈希值，再将两者进行比较来验证签名的真实性。

示例 20-2 假设待签名的消息 $m=14$，签名者选择的素数 $p=23$，选择的素数 p 的本原根 $g=11$，并根据私钥选择条件选择了私钥 $x=7$，并选择了秘密数 $k=17$ 用于签名。试描述签名和验证的具体过程。

解 根据条件有 $p=23, g=11, x=7, k=17, m=14$

(1) 计算公钥

$$y = g^x \bmod p = 11^7 \bmod 23 = 7$$

签名者发布公钥$(p,g,y)=(23,11,7)$用于验证签名。

(2) 签名

根据条件计算 $a = g^k \bmod p = 11^{17} \bmod 23 = 14$

根据 $s=k^{-1}(m-ax) \bmod (p-1)$ 计算签名值，由于 k^{-1} 是 $k \bmod (p-1)$ 的乘法逆元，因此可以计算得到 $k^{-1}=13$，则 $s=13(14-14\times7) \bmod 22=8$，得到 $(a,s)=(14,8)$ 并发送给接收方。

(3) 验证

接收方在收到签名后通过 $y^a a^s \bmod p=g^m \bmod p$ 来验证。

$$y^a a^s \bmod p = 7^{14} \times 14^8 \bmod 23 = 3$$
$$g^m \bmod p = 11^{14} \bmod 23 = 3$$

接收方验证签名无误。

20.4 DSA 数字签名方案

DSA(Digital Signature Algorithm)数字签名方案是美国联邦信息处理标准(FIPS PUB 186)推荐的数字签名方案。目前 DSA 签名算法在 IEEE 的 P1363 标准中的数字签名方案中也被推荐使用。

DSA 数字签名方案的安全性是基于有限域上计算离散对数的困难性问题，DSA 签名是针对待签名消息的散列值，相应散列值的计算使用的是 SHA-1 算法。

DSA 数字签名方案的签名与验证的基本过程如图 20-5 所示。

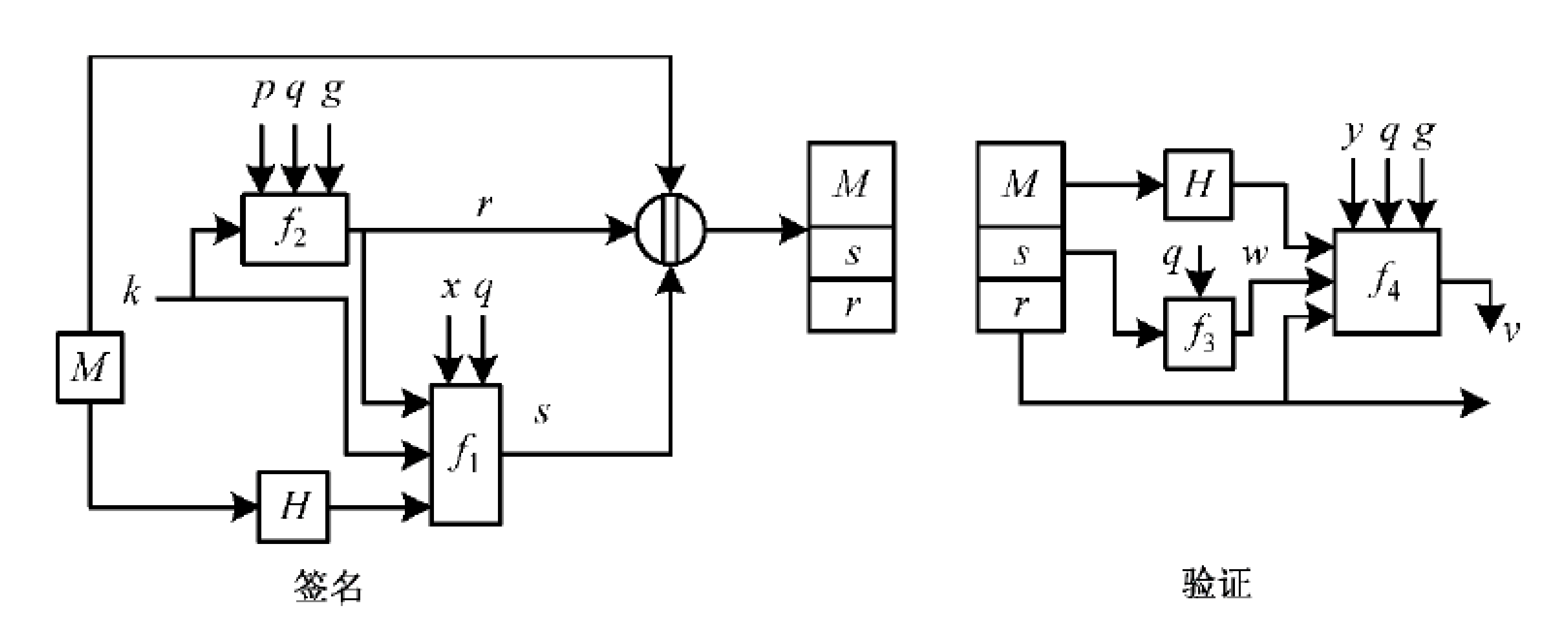


图 20-5 DSA 数字签名方案示意图

DSA 数字签名方案的签名算法为

$$\begin{aligned} r &= f_2(k,p,q,g) = (g^k \bmod p) \bmod q \\ s &= f_1(H(M),k,x,r,q) = k^{-1}(H(M)+xr) \bmod q \end{aligned} \tag{20-4}$$

DSA 数字签名方案的验证算法为

$$\begin{aligned} w &= f_3(s,q) = (s^{-1} \bmod q) \\ u_1 &= (H(M)^w) \bmod q \\ u_2 &= r^w \bmod q \\ v &= ((g^{u_1} y^{u_2}) \bmod p) \bmod q \end{aligned} \tag{20-5}$$

DSA 算法的要求与具体签名和验证过程如下：

(1) DSA 参数

① 参数 p 是一个素数，满足在 $512\leqslant L\leqslant1024$ 下 $2^{L-1}<p<2^L$，并且 L 为 64 的倍数，即 p 的长度在 512 位至 1024 位之间，长度的增量是 64。

② 参数 q 是 $p-1$ 的一个素因子，q 的长度为 160 位，即 $2^{159}<q<2^{160}$。

③ 参数 $g=h^{(p-1)/q} \bmod p$，其中参数 h 是满足 $1<h<p-1$ 的任意整数，并且满足 $h^{(p-1)/q} \bmod p>1$。

④ 私钥 x 是用户选择的随机整数，并要求 $0<x<q$。

⑤ 公钥 $y=g^x \bmod p$。对于给定的 x 计算 y 是容易的，而对于给定的 y 计算 x 是困难的，这就是 DSA 数字签名方案的安全基础。

参数 p,q,g 是公开的系统参数，参数 x 是用户的私钥，参数 y 是用户的公钥。

(2) 签名过程

假设用户 A 要对消息 M 进行签名，那么具体签名过程如下：

① 首先用户 A 选择一个秘密的随机整数 k，满足 $1\leqslant k\leqslant p-2$，同时 k 与 $p-1$ 互素。

② 用户 A 计算

$$\begin{aligned} r &= (g^k \bmod p) \bmod q \\ s &= k^{-1}(H(M)+xr) \bmod q \end{aligned} \tag{20-6}$$

计算得到的(r,s)是对消息 M 的数字签名。

(3) 验证过程

用户 B 在收到用户 A 对消息 M 的签名之后，通过以下步骤进行验证：

① 计算 w，w 的计算方法如下：

$$w = s^{-1} \bmod q \tag{20-7}$$

② 计算 u_1 和 u_2，u_1 和 u_2 的计算方法如下：

$$\begin{aligned} u_1 &= (H(M)^w) \bmod q \\ u_2 &= r^w \bmod q \end{aligned} \tag{20-8}$$

③. 计算 v，v 的计算方法如下：

$$v = ((g^{u_1} y^{u_2}) \bmod p) \bmod q \tag{20-9}$$

④ 验证 $v=w$ 是否成立，若成立则签名为真，否则签名为假。

示例 20-3 假设用户 A 为签名者，用户 A 选择素数 $q=11$，$p=2\times 11+1=23$，选择 $h=7$，符合 $1<h<p-1$，计算 $g=h^{(p-1)/q} \bmod p=7^{(23-1)/11} \bmod p=3$，满足 $g>1$。用户 A 随机选择私钥 $x=5$，满足 $0<x<q$，计算 $y=g^x \bmod p=3^5 \bmod 23=13$。

公开的参数为$(p,q,g)=(23,11,13)$，公钥为 $y=13$，私钥为 $x=5$，假设待签名的消息为 $H(M)=6$，试给出签名和验证过程。

解 用户 A 选择 $k=9$，满足与 $p-1=22$ 互素。

用户 A 的签名过程如下：

计算：$r=(g^k \bmod p) \bmod q=(139 \bmod 23) \bmod 11=3$。

计算：k^{-1}，由 $k^{-1}k=1 \bmod q$ 计算得到 $k^{-1}=5$。

计算：$s=k^{-1}(H(M)+xr) \bmod q=5\times(6+5\times 3) \bmod 11=6$。

将签名结果$(r,s)=(3,6)$发送给用户 B。

用户 B 的验证过程如下：

计算：$w=s^{-1} \bmod q=2 \bmod 11=2$。

计算：$u_1=(H(M)^w) \bmod q=6^2 \bmod 11=3$。

计算：$u_2=r^w \bmod q=3^2 \bmod 11=9$。

计算：

$$v=((g^{u_1}y^{u_2})\bmod p)\bmod q=((13^3\times 13^9)\bmod 23)\bmod 11=2$$

结论：因为 $v=w$，所以签名正确。

20.5 盲签名

通常在进行数字签名时，签名者是知道所要签名的内容。但有时候所需签名的内容并不想让签名者知道具体内容，此时就需要使用盲签名。例如，在进行电子投票时，通常投票者不希望签名者知道投票的内容，此时则可以通过盲签名来完成签名。又如，在进行电子购物时，购物者需要银行进行签名，但同样也不希望银行知道购物的内容，此时也需要使用盲签名来完成签名。

20.5.1 盲签名基本原理

盲签名最早在 1982 年提出，其目的是有效保护待签署消息的内容。

盲签名首先由消息提供者对消息进行盲化处理，然后发送给签名者，签名者对盲化的消息进行签名。在消息拥有者得到签名后的消息之后，对签名的盲化消息进行去盲，得到签名者对于原消息的签名。盲签名是一种特殊的数字签名方式，在签名过程中签名者并不知道签名的具体内容。

盲化签名技术除了要满足一般签名的要求外，还需要满足以下条件：

(1) 签名者对所签署的消息是不可见的，即签名者无法知道他所签名的内容。

(2) 签名后的消息是不可追踪的，即但签名的消息被公布后，签名者无法知道这是哪次的签名。

盲签名实际上用到两个算法，第一个是加密算法，使用加密算法可以达到对消息的隐藏，实现了盲化的目的，第二个算法是签名算法，实现了对盲化消息的签名。

一个好的盲签名算法需要具有以下性质：

(1) 不可伪造性——除了签名者本人，任何其他人都不能以他的名义生成有效的签名。

(2) 不可抵赖性——签名者一旦签署了某个消息，那么他就无法否认对消息的签名。

(3) 盲性——签名者虽然对某个消息进行了签名，但他却无法看到消息的内容。

(4) 不可跟踪性——一旦签名的消息被公开后，签名者无法确定自己是何时进行的签名。

以上 4 条性质是设计盲签名时需要遵循的基本标准，满足这些性质的盲签名方案被认为是安全的盲签名方案。

盲签名的基本过程可以描述如下：

(1) 用户 A 使用加密算法对需签名的消息进行加密：

$$C=E_{ka}(M)$$

将加密后的消息发送给签名者 B。

(2) 签名者 B 使用签名算法及密钥对盲化消息 C 进行签名：

$$S_c=\mathrm{Sig}_{kb}(C)$$

并将 S_c 发送给用户 A。

(3) 用户 A 对签名的消息进行解盲：

$$S_M = D_{ka}(S_c) = D_{ka}(\mathrm{Sig}_{kb}(C)) = \mathrm{Sig}_{kb}(D_{ka}(E_{ka}(M))) = \mathrm{Sig}_{kb}(M)$$

这样用户 A 便得到了签名者 B 对消息 M 的签名。

盲签名的具体算法和实施过程根据所选定的签名算法而定，并且要求盲化处理过程和签名算法必须可换，否则无法得到最终的签名。

20.5.2 RSA 盲签名

RSA 盲签名算法是 David Chaum 利用 RSA 算法设计的第一个盲数字签名方案。RSA 签名方案首先也是对需进行签名的消息进行盲化，然后对盲化的消息进行签名，再进行脱盲得到签名的消息。

假设 RSA 盲签名的公钥 (e,n)，私钥是 (d,n)，待签名的消息是 m，那么，RSA 盲签名方案的基本过程可以描述如下：

(1) 用户 A 选择一个随机数 r，使 r 满足 $\gcd(r,n)=1$，即 r 与 n 互素。

(2) 用户 A 计算：

$$m' \equiv mr^e \bmod n$$

并将 m' 发送给签名者 B。

(3) 签名者 B 对得到的 m' 进行签名，签名方法如下：

$$s' = (m')^d \bmod n$$

并将签名结果发送给用户 A。

(4) 用户 A 在得到签名者 B 的签名后进行脱盲，脱盲方法如下：

$$s \equiv s' \cdot r^{-1} \bmod n$$

由此，用户 A 得到了签名者 B 对消息 m 的签名。

RSA 盲签名算法利用了 RSA 公钥算法的基本特性，即 $r^{ed} \equiv r \bmod n$，得到签名的基本过程为

$$s \equiv s' \cdot r^{-1} \equiv (m')^d r^{-1} \equiv m^d r^{ed} r^{-1} \equiv m^d r r^{-1} \equiv m^d \bmod n$$

在 RSA 盲签名过程中由于 r 是随机选择的，因此，用户 A 对消息进行盲化得到的盲化值也是随机的，所以，签名者无法得知所签名的消息的具体内容。同时，RSA 盲签名算法仅使用了一对公钥和私钥便完成盲化、签名和脱盲的过程。

20.6 习题与实践题

20.6.1 习题

1. 简要说明数字签名的基本过程。

2. 简要说明 RSA 签名方案的签名过程。

3. 设 $p=23$，$q=11$，待签名的消息 $M=27$，假设采用的签名方案是 RSA 签名方案，试给出签名和验证的计算过程。

4. 简要说明 Elgamal 签名方案的签名过程。

5. 假设待签名的消息 $m=17$,并选择 Elgamal 签名方案进行签名,签名者选择的素数 $p=23$,选择的素数 p 的本原根 $g=11$,并根据私钥选择条件选择了私钥 $x=9$,并选择了秘密数 $k=7$ 用于签名。试描述签名和验证的具体过程。

6. 简要说明 DSA 签名方案的基本过程。

7. 简要说明 RSA 盲签名的基本原理。

20.6.2 实践题

1. 在整型数据大小的范围内模拟实现 RSA 数字签名方案。

2. 在整型数据大小的范围内模拟实现 Elgamal 数字签名方案。

3. 在整型数据大小的范围内模拟实现 RSA 盲签名方案。

参 考 文 献

[1] 傅祖芸. 信息论-基础理论与应用[M]. 北京：电子工业出版社，2001.
[2] 王昭，袁春. 信息安全原理与应用[M]. 北京：电子工业出版社，2012.
[3] William Stalling. 密码编码学与网络安全-原理与实践[M]. 刘玉珍，译. 北京：电子工业出版社，2004.
[4] Thomas H. Cormen，Charles E. Leiserson，等. 算法导论[M]. 潘金贵，译. 北京：机械工业出版社，2006.
[5] 同济大学数学系. 线性代数[M]. 北京：清华大学出版社，2007.
[6] 刘家勇. 应用密码学[M]. 北京：清华大学出版社，2008.
[7] Pallab Ghosh. 数值方法(C++ 描述)[M]. 徐士良，译. 北京：清华大学出版社，2008.
[8] Vimalathithan. R，Dr. M. L. Valarmathi. Cryptanalysis of S-DES using Genetic Algorithm[J]. International Journal of Recent Trends in Engineering，2009，4：76-79.
[9] Bhanu Arya，Abrar Ahmad Berqi. Implementing CBC on S-DES using GA[C]. 2nd National Conference in Intelligent Computing & Communication. Organized by Dept. of IT，GCET Greater Noida，INDIA，2011.
[10] Feistel cipher. http：//en. wikipedia. org/wiki/Feistel_cipher.
[11] 徐茂智. 信息安全基础[M]. 北京：高等教育出版社，2006.
[12] 郑东，李祥学，黄征. 密码学——密码协议与算法[M]. 北京：电子工业出版社，2009.
[13] Announcing the Advanced Encryption Standard(AES)[S]. Federal Information Processing Standards Publication 197. 2001.
[14] Bruce Schneier. 应用密码学-协议、算法与 C 源程序[M]. 吴世忠，祝世雄，张文政，等，译. 北京：机械工业出版社，2000.
[15] 陈鲁生，沈世镒. 现代密码学[M]. 北京：科学出版社，2008.
[16] 杨晓元，魏立线. 计算机密码学[M]. 西安：西安交通大学出版社，2007.
[17] Charles P. Pfleeger，Shari Lawrence Pfleeger. 信息安全原理与应用[M]. 李毅超，蔡洪斌，谭浩，译. 北京：电子工业出版社，2007.
[18] Christof Paar，Jan Pelzl. 深入浅出密码学-常用加密技术原理与应用[M]. 马小婷，译. 北京：清华大学出版社，2012.
[19] 张青凤，殷肖川，李长青. IDEA 算法及其编程实现[J]. 现代电子技术，2006(1)：69-71.
[20] Oleg Vyshnyvetskyl，Sebastien Guilloux. IDEA Block Cipher Final Report. http：// people . rit. edu/sxg1678/cryptography/report. pdf.
[21] Willi Meier. On the Security of the IDEA Block Cipher. Advances in Cryptology—EUROCRYPT'93[C]. LNCS765：371-385.
[22] Bruce Schneier. Description of a New Variable—Length Key，64-Bit Block Cipher. http：//www. schneier. com/paper-blowfish-fse. html.
[23] Nick Hoffman. A Simplified Idea Algorithm[J]. Cryptologia，2007，31(2)：143-151.
[24] 钟黔川，朱清新. Blowfish 密码系统分析[J]. 计算机应用，2007，27(12)：2940-2942.
[25] Gurjeevan Singh，Ashwani Kumar，K. S. Sandha. A Study of New Trends in Blowfish Algorithm

[J]. Journal of Engineering Research and Applications (IJERA),2011,1(2):321-326.

[26] Krishnamurthy G. N,Dr. V. Ramaswamy,Leela G. H, et al. Blow-CAST-Fish: A New 64-bit Block Cipher[J]. International Journal of Computer Science and Network Security,2008,8(4): 282-290.

[27] Krishnamurthy G. N,Dr. V. Ramaswamy,Leela G. H, et al. Performance enhancement of Blowfish and CAST-128 algorithm and Security analysis of improved Blowfish algorithm using Avalanche effect[J]. International Journal of Computer Science and Network Security,2008,8(3): 244-250.

[28] C. Adams. The CAST-128 Encryption Algorithm. http://tools.ietf.org/html/rfc2144.

[29] J. Altman. Telnet Encryption: CAST-128 64 bit Output Feedback. https://www.ietf.org/rfc/rfc2949.txt.

[30] Block cipher mode of operation. http://en.wikipedia.org/wiki/Block_cipher_mode_of_ operation# Cipher_feedback_.28CFB.29.

[31] Padding. http://en.wikipedia.org/wiki/Padding_(cryptography).

[32] C. Paar,J. Pelzl. Understanding Cryptograph[M]. Berlin Heidelberg: Springer-Verlag,2010.

[33] Alex Biryukov,Adi Shamir,David Wagner. Real Time Cryptanalysis of A5/1 on a PC[C]. Fast Software Encryption Lecture Notes in Computer Science,7th International Workshop,FSE 2000 New York,NY,USA,2001:1-18.

[34] Eli Biham,Orr Dunkelman. Cryptanalysis of the A5/1 GSM Stream Cipher[C]. Progress in Cryptology—INDOCRYPT 2000 Lecture Notes in Computer Science,First International Conference in Cryptology in India Calcutta,India. 2000:43-51.

[35] Alpesh R. Sankaliya,V. Mishra,Abhilash Mandloi. Implementation of Cryptographic Algorithms for GSM Cellular Standard[J]. Ganpat University Journal of Engineering & Technology,2011,1: 14-18.

[36] A5/1. http://en.wikipedia.org/wiki/A5/1.

[37] Moti Yung. Random Shuffles of RC4[C]. Advances in Cryptology—CRYPTO 2002,22nd Annual International Cryptology Conference Santa Barbara,California,USA,2002:18-22.

[38] RC4. http://en.wikipedia.org/wiki/RC4.

[39] Orr Dunkelman. Key Collisions of the RC4 Stream Cipher[C]. Fast Software Encryption,16th International Workshop,FSE 2009 Leuven,Belgium,February 22-25,2009: 38-50.

[40] Mitsuru Matsui. A Practical Attack on Broadcast RC4[C]. Fast Software Encryption,8th International Workshop,FSE 2001 Yokohama,Japan,April 2-4,2001:152-164.

[41] Ronald L. Rivest. The RC5 encryption algorithm[C]. Fast Software Encryption,Second International Workshop Leuven,Belgium,December 14-16,1994:86-96.

[42] Henk C. A. van Tilborg,Sushil Jajodia. RC5[C]. Encyclopedia of Cryptography and Security,2011:1032-1033.

[43] Alex Biryukov,Eyal Kushilevitz. Improved cryptanalysis of RC5[C]. Advances in Cryptology—EUROCRYPT'98 Lecture Notes in Computer Science Volume 1403,1998:85-99.

[44] Stafford Tavares,Henk Meijer. A Timing Attack on RC5[C]. 5th Annual International Workshop,SAC'98 Kingston,Ontario,Canada,August 17-18,1998:306-318.

[45] Henk C. A. van Tilborg. Encyclopedia of Cryptography and Security[M]. Springer US,2005:516.

[46] Henk C. A. van Tilborg,Sushil Jajodia . Encyclopedia of Cryptography and Security[M]. Springer US,2011:1033-1034.

[47] Ronald L. Rivest,M. J. B. Robshaw,R. Sidney. The RC6 Block Cipher. ftp://cs.usu.edu.ru/crypto /RC6/rc6v11.pdf.

[48] Jean-Luc Beuchat. FPGA Implementations of the RC6 Block Cipher[C]. Field Programmable Logic and Application, Lecture Notes in Computer Science Volume 2778, 2003: 101-110.
[49] 多项式时间. http://zh.wikipedia.org/wiki/多项式时间.
[50] Miller-Rabin primality test. http://en.wikipedia.org/wiki/Miller-Rabin_primality_test.
[51] Solovay-Strassen primality test. http://en.wikipedia.org/wiki/Solovay-Strassen_primality_test.
[52] 陈恭亮. 信息安全数学基础[M]. 北京：清华大学出版社，2004.
[53] Lehmann primality test. http://en.algoritmy.net/article/48610/Lehmann-test.
[54] AKS primality test. http://en.wikipedia.org/wiki/AKS_primality_test.
[55] Manindra Agrawal, Neeraj Kayal, Nitin Saxena. PRIMES Is in P[J]. Annals of Mathematics, 2004, 160(2): 781-793.
[56] RSA (cryptosystem). http://en.wikipedia.org/wiki/RSA_(cryptosystem).
[57] Digital signature. http://en.wikipedia.org/wiki/Digital_signature.
[58] Diffie-Hellman key exchange. http://en.wikipedia.org/wiki /Diffie-Hellman_key_exchange.
[59] General number field sieve. http://en.wikipedia.org/wiki/General_number_field_sieve.
[60] Thorsten Kleinjung. On polynomial selection for the general number field sieve[J]. Mathematics of computation, 2006, 75(256): 2037-2047.
[61] Elgamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms[J]. Information Theory, IEEE Transactions on, 1985, 31(4): 469-472.
[62] Benoit Chevallier-Mames, Pascal Paillier, David Pointcheval. Encoding-Free ElGamal Encryption Without Random Oracles[C]. Public Key Cryptography - PKC 2006. 2006: 91-104.
[63] Mike Burmester, Yvo Desmedt, Hiroshi Doi. A Structured ElGamal-Type Multisignature Scheme [C]. Public Key Cryptography - PKC 2000. 2000: 466-483.
[64] ElGamal encryption. http://en.wikipedia.org/wiki/ElGamal.
[65] Cryptographic hash function. http://en.wikipedia.org/wiki/Cryptographic_hash_function.
[66] Ronald L. Rivest. The MD4 Message Digest Algorithm[C]. Advances in Cryptology-CRYPT0'90. 1991: 303-311.
[67] MD4. http://en.wikipedia.org/wiki/MD4.
[68] MD5. http://en.wikipedia.org/wiki/MD5.
[69] The MD5 Message-Digest Algorithm. http://tools.ietf.org/html/rfc1321.
[70] SHA-1. http://en.wikipedia.org/wiki/SHA1.
[71] US Secure Hash Algorithm 1 (SHA1). http://tools.ietf.org/html/draft-eastlake-sha1-01.
[72] Digital signature. http://en.wikipedia.org/wiki/Digital_signature.
[73] Dan Boneh. Encyclopedia of Cryptography and Security[M]. Springer US, 2011.
[74] ElGamal signature scheme. http://en.wikipedia.org/wiki/ElGamal_signature_scheme.
[75] RIPEMD. http://en.wikipedia.org/wiki/RIPEMD.
[76] The hash function RIPEMD-160. http://homes.esat.kuleuven.be/~bosselae/ripemd160.html.
[77] Hans Dobbertin, Antoon Bosselaers, Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD[C]. Fast Software Encryption. Lecture Notes in Computer Science Vol 1039, 1996: 71-82.
[78] Xuejia Lai, Moti Yung, Dongdai Lin. Preimage Attacks on Step-Reduced RIPEMD-128 and RIPEMD-160[C]. Information Security and Cryptology, 6th International Conference, Inscrypt 2010, Shanghai, China, October 20-24, 2010, Revised Selected Papers, 2010: 169-186.

[79] Kazue Sako, Palash Sarkar. Improved Cryptanalysis of Reduced RIPEMD-160[C]. Advances in Cryptology—ASIACRYPT 2013, 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part Ⅱ, 2013: 484-503.

[80] Digital Signature Algorithm. http://en.wikipedia.org/wiki/Digital_Signature_Algorithm.

[81] Blind signature. http://en.wikipedia.org/wiki/Blind_signature.